FreeBSD™ JOURNAL

Nov/Dec 2016

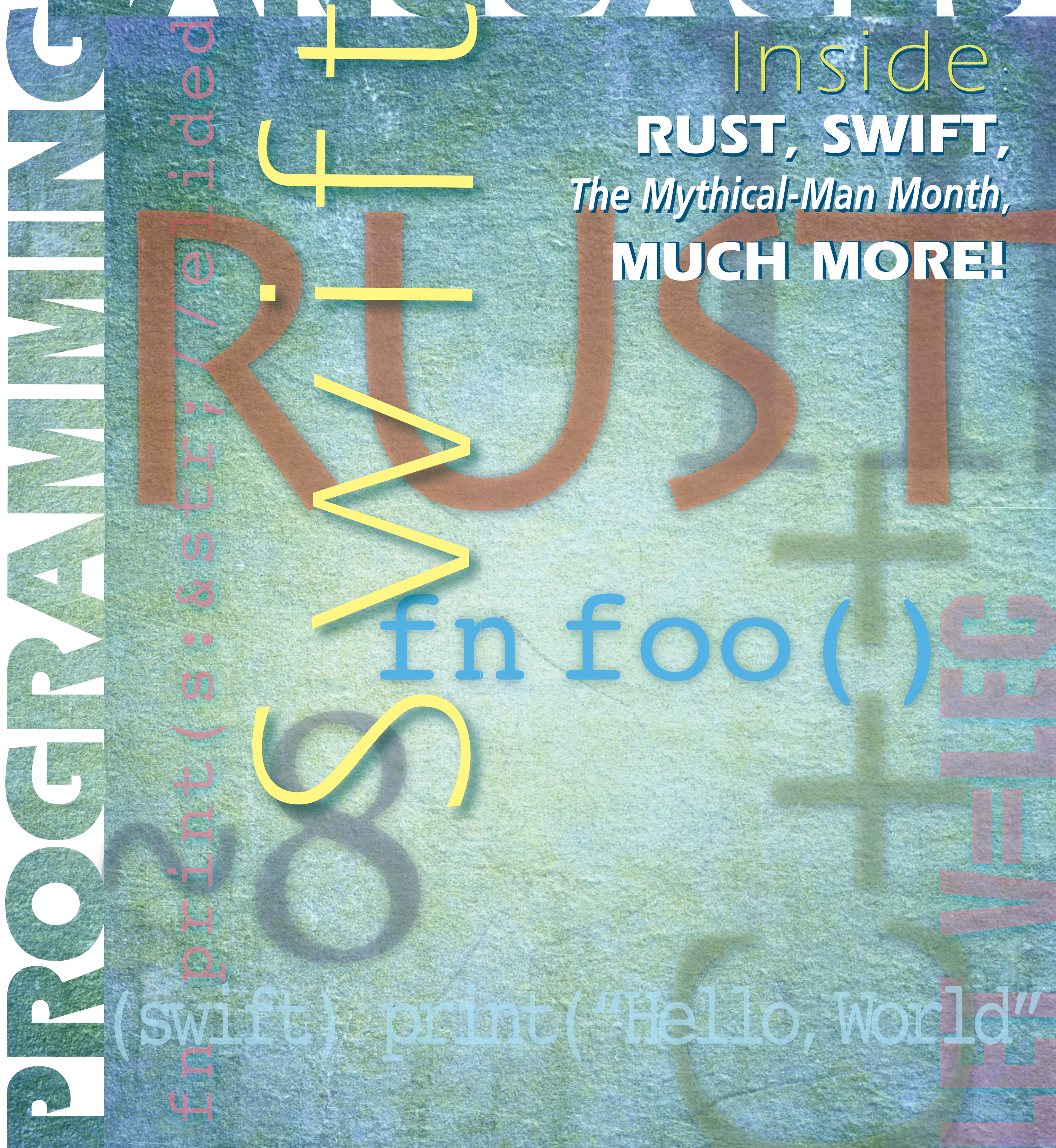
PROGRAMMING LANGUAGES

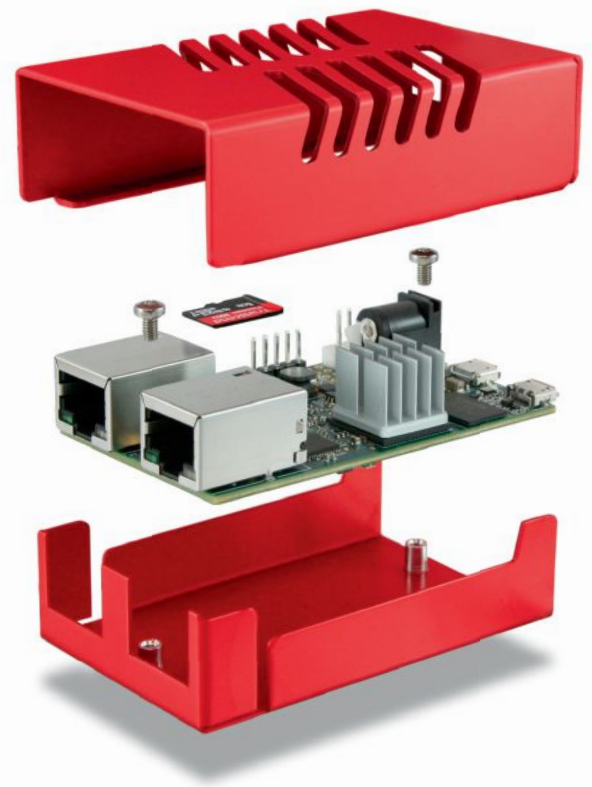
Inside:

RUST, SWIFT,

The Mythical-Man Month,

MUCH MORE!





INTRODUCING THE

SG-1000

microFirewall

Shipping Now!

$149

Includes pfSense® Gold, a $99 value.

pfSense®

You asked. We delivered! The new Netgate® SG-1000 microFirewall is a cost-effective, state-of-the-art, ARM®-based, pfSense Security Gateway appliance. The SG-1000 comes with dual 1Gbps Ethernet ports, enabling maximum throughput exceeding 300 Mbps. The ARM Cortex®-A8 in the TI AM3552 SoC and DDR3L RAM combine to facilitate low power consumption while maintaining performance. The SG-1000 comes in a lightweight and durable anodized aluminum case. Its credit-card sized form-factor allows it to be easily tucked away, but you'll be proud to show it off.

- The ideal security endpoint for the Internet of Things (IoT)
- SCADA firewall for infrastructure & commercial systems
- Secure multi-dwelling units (MDU) such as dorm rooms & apartments
- Deploy as an IPMI port firewall to close critical IPMI / BMC vulnerabilities
- Portable VPN endpoint firewall, network tap or on-premises SMB appliance
- 5 V DC power for a wide range of applications with low energy consumption

Netgate®

Shop now at the Netgate store or authorized partners worldwide.
www.netgate.com/products/sg-1000.html

Table of Contents

FreeBSD JOURNAL

Nov/Dec 2016

Simplify your Data Center

Meet TrueRack™ — A powerfully flexible rack-scale architecture that takes the guesswork out of building large scale data center applications.

- **Converged Infrastructure**
- **Customizable for Virtualization, Big Data, Cloud & Hyperscale**
- **Up to 70% Lower TCO Than AWS**
- **Scalable and Repeatable Deployments**

For more information on TrueRack, visit **iXsystems.com/TrueRack** today

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Copy Editor • Annaliese Jakimedis

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Office Administrator • Michael Davis
davism@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
5757 Central Ave., Suite 201, Boulder, CO 80301
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org
Copyright © 2016 by FreeBSD Foundation. All rights reserved.

This magazine may not be reproduced in whole or in part without written permission from the publisher.

Outreach and Support

It's that time of the year when the FreeBSD Foundation, sponsor of the *FreeBSD Journal*, asks people to donate to our end-of-year fundraising drive. As readers know, the FreeBSD Foundation supports all aspects of the FreeBSD Project, not just this *Journal*. In 2016 we have sponsored BSD conferences, including AsiaBSDCon, BSDCan, and EuroBSD, and we have undertaken significant outreach at non-BSD events so that we not only reach people who already know about FreeBSD, but also those who should know more about FreeBSD. Speaking of outreach, I'd like to encourage each and every one of this *Journal*'s readers to show this issue to their friends and colleagues, and encourage them to subscribe as well, which is just another way to grow the FreeBSD Project.

Continuing the outreach topic, in this issue George Rosemond, from NYC*BUG (New York City's BSD User Group) writes about how to start a BSD user group. User groups are one of the foundations of open-source projects. Through meetings, talks, and local conferences, they bring new people into contact with others who work on the same software. If you've ever considered starting a BSD user group, this is a piece you won't want to miss.

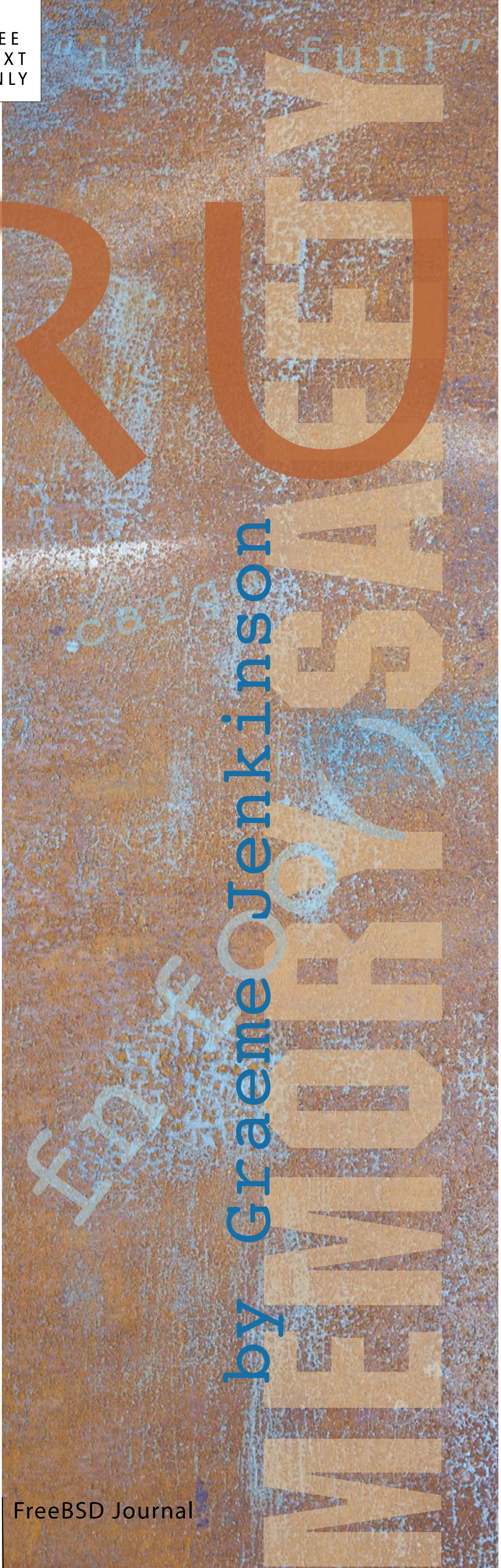
FreeBSD continues to be a platform of choice for new systems. Two of the latest languages, Rust and Swift, are covered in this issue. Bringing new users to FreeBSD also means education, and also in this issue is an article Benedict Reuschling and I wrote about our experience teaching computer science undergraduates with FreeBSD. (All the material for our course is available under a BSD license at teachbsd.org.)

The book review this month is of a very special, and relatively short, volume. For over 30 years, Fred Brooks's book *The Mythical Man-Month* has remained a stalwart of software engineering. For this issue, John Baldwin reviews Brooks's classic work with an eye toward applying its lessons to FreeBSD.

In closing, I want to make one final appeal to support the FreeBSD Foundation. Helping the Foundation meet and exceed this year's fundraising goals ensures that the FreeBSD Project will continue to be successful in 2017 and beyond.

Thank you. George Neville-Neil
Editor in Chief of the *FreeBSD Journal*,
Director of the FreeBSD Foundation

SEE TEXT ONLY



IT'S BETTER TO RUST THAN WEAR OUT

by Graeme Jenkinson

When a colleague of mine first enthused to me about Rust, I was skeptical. Back in the day, I'd cut my programming teeth developing software for safety-critical systems, and I'd learned the hard way that programming languages are frequently less sane than they first appear. Take C. Despite a considerable standardization effort, the C specification remains riddled with unspecified, undefined, and implementation-defined behaviors [2]. And even in 2016, researchers continue to explore the differences between the C ISO standard and the de facto usage [4]. While not all software engineers need be concerned with the seemingly esoteric issues of what happens when a bit field is declared with a type other than `int`, `signed int` or `unsigned int` (it's undefined [2]), I'd worked too long with safety-critical and security systems to switch off this retentive part of my brain. And so, somewhat dismissively, I mentally parked Rust along with Go, Haskell, and all the other technologies that sound cool, but I could never foresee actually using. Then early this year, I had the opportunity to revisit Rust, and I found I'd been a bit hasty.

I had been developing a prototype for a distributed tracing framework built on top of DTrace. The prototype, written in C, acted as a DTrace consumer (interfacing with `libdtrace`) and sent DTrace records upstream for further processing (aggregation, reordering, and so on) using Apache Kafka. For a prototype this worked fine, but as the work progressed, I needed to rapidly explore the design space. This task favored adopting higher-level language, but which one to choose? Like all good engineers, I started to list out my requirements. I needed a language that emphasized programmer productivity. It needed to easily and efficiently interface with libraries written in C (such as `libdtrace`). I also needed easy deployment, therefore languages requiring a heavy runtime (and Java specifically) were complete nonstarters. Good support for concurrency and, ideally, prevention of data races would be nice. And finally, with my security hat on, I didn't want to embarrass myself by introducing a bucket-load of exploitable vulnerabilities. I thought back to that earlier conversation with my colleague, aren't these requirements exactly what Rust is designed for? And so I decided to give Rust a whirl, and I'm glad that I did, because I really liked what I found. So what's Rust all about?

Rust's vision is simple—to provide a safe alternative to C++ that makes system programmers more productive, mission-critical software less prone to bugs, and parallel algorithms more tractable. Rust's main benefits are [5]:

- Zero-cost abstractions,
- guaranteed memory safety (without garbage collection),
- threads without data races,
- type inference,
- minimal runtime, and
- efficient C bindings.

The Rust language has a number of comprehensive tutorials, notably the "Rust Book" [5]. Therefore, rather than retreading that ground, I will instead highlight the features of Rust that I find particularly compelling. Along the way, I'll discuss the features of Rust that are most difficult to master. And finally, I'll show how to get started programming in Rust on FreeBSD.

FIGHTING THE BORROW CHECKER

Before diving in headfirst and firing up your favorite text editor (vim, obviously), it is important to understand Rust's most significant cost, its steep learning curve. On that learning curve, nothing is more frustrating than repeatedly invoking the wrath of the "borrow checker" (the notional enforcer of Rust's ownership system). Ownership is one of Rust's most compelling features, and it provides the foundations on which Rust's guarantees of memory safety are built. In Rust, a variable binding (the binding of a value to a name) has ownership of the value it is bound to. Ownership is mutually exclusive; that is, a resource must have a single owner. It is the borrow checker's job to enforce this invariant, which it does by failing early (at compile time) and loudly.

In the following example, taken from the "Rust Book" (*The Rust Programming Language*, 2016), `v` bound to the vector `vec![1, 2, 3]` (`vec![1, 2, 3]` is a Rust macro creating a contiguous, growable array containing the values 1, 2 and 3). The function `foo()` is the "owning scope" for variable binding `v`. When `v` comes into scope, a new vector is allocated on the stack and its elements on the heap; when the scope ends, `v`'s memory (both the components on the stack and on the heap) is automatically freed. Yay, memory safety without garbage collection.

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

Ownership can be transferred through an assignment `let x = y` (move semantics). But remember ownership is mutually exclusive, so in the example below, when the variable `v` is referenced (in the `println` macro) after the transfer of ownership to `v2`, the borrow checker cries foul (`error: use of moved value:` \``v`\`).

```
let v = vec![1, 2, 3];
let v2 = v;

println!("v[0] is: {}", v[0]);
```

In the example below, calling the function `bar()` passing the vector `v` as an argument transfers the ownership of `v`. When the owning scope (the function bar) ends, `v`'s memory is automatically freed (as before). Ownership of `v` can be returned to the caller by simply returning `v` from `bar`. This approach would get tedious pretty quickly, and so Rust allows borrowing of a reference (that is, "borrowing" the ownership of the variable binding). A borrowed binding does not deallocate the resource when the binding goes out of scope. This means that after the call to `bar()`, we can use our original bindings once again (see following page):

```
fn bar(v: &Vec<i32>) {
    // do something useful v here
}

let v = vec![1, 2, 3];

bar(&v);

println!("v[0] is: {}", v[0]);
```

IMMUTABILITY BY DEFAULT

By default, Rust variable bindings are immutable. Having spent many an hour typing `const * const` and `final` (in C and Java, respectively) this feature alone fills me with joy; and what is more, unlike `const`, it actually provides immutability. Variable bindings can be specified as mutable using the `mut` keyword: `let mut x = 10` (also note the sensible use of type inference). Like variable bindings, references are immutable by default and can be made mutable by the addition of the `mut` keyword (`&mut T`). Shared mutable state causes data races. Rust prevents shared mutable state by enforcing that there is either:

- one or more references (`&T`) to a resource or
- exactly one mutable reference (`&mut T`).

CHOOSING YOUR GUARANTEES

Rust's philosophy is to provide the programmer with control over guarantees and costs. Rust's rule that there can be one or more immutable references or exactly one mutable reference is enforced at compile time. However, in keeping with the overall philosophy, various different trade-offs between runtime and compile time enforcement are supported. A reference counted pointer (`Rc<T>`) allows multiple "owning" pointers to the same (immutable) data; the data is dropped and memory freed only when all the referenced counter pointers are out of scope. This is useful when read-only data is shared and it is non-deterministic to when all consumers have finished accessing the data. A reference counted pointer gives a different guarantee (that memory is freed when all owned pointers go out of scope) than the compile time enforced guarantees of the ownership system. However, this comes with additional costs (memory and computation to maintain the reference count). Similarly, mutable state can be shared (using a `Cell<T>` type); this again brings different trade-offs for guarantees and costs.

LIFETIMES

There is one final and rather subtle issue with ownership. Variable bindings exist within their owned scope, and borrowed references to these bindings also exist within their own separate scope. When variable bindings go out of scope, the ownership is relinquished and the memory is automatically freed. So what would happen if a variable binding went out of scope while a borrowed reference was still in use? In summary, really bad things invalidate Rust's guarantees of memory safety. Therefore, this can't be allowed to happen. Lifetimes are Rust's mechanism to prevent borrowed references from outliving the original resource.

In Rust, every reference has an associated lifetime. However, lifetimes can often be elided. The example below shows equivalent syntax with the lifetime (`'a`) of the reference `s` elided and made explicit:

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded
```

Global variables are likely to be the novice Rust programmer's first interaction with lifetimes. Global variables are specified with Rust's special static lifetime as follows `static N: i32 = 5`;. A static lifetime specifies that the variable binding has the lifetime of the entire program (note that string literals possess the type `&'static str`, and therefore live for the entire life of the program). If I were to hazard a guess at where lifetimes next rear their heads, it would be storing a reference in a `struct`. In Rust, a `struct` is used to create complex (composite) datatypes. When Rust structs contain references (that is, they borrow ownership), it is important to ensure that any references to the `struct` do not outlive any references that the `struct` possesses. Therefore, a Rust `struct`'s lifetime must be equal to or shorter than that of any references it contains.

EFFICIENT INHERITANCE

In contrast to C++ and Java's heavyweight approach to inheritance, Rust takes a muted approach, and, in fact, the word inheritance is studiously avoided. With traditional inheritance gone, AWOL classes are no longer needed. Having been freed from confines of classes, methods can be defined anywhere and types can have an arbitrary collection of methods. As in Go, inheritance in Rust has been boiled down to simply sharing a collection of method signatures (this approach is sometimes referred to as objects without classes). Rust Traits group together a collection of methods signatures—a Rust type can implement an arbitrary set of Traits (thus Traits are similar to `mixins`).

FIGHTING THE BORROW CHECKER REDUX

What makes Rust's ownership system so tricky to master? Ownership is not a complexity introduced by the Rust language; it is an intrinsic complexity of programming regardless of the language being used. Languages that fail to address ownership fail at runtime (with data races and so on). In contrast, Rust makes issues of ownership explicit, allowing the language to fail early and loudly at compile time. Rust's borrow checker is like that friend you couldn't quite get on with on first meeting. Over time, and once they've helped you out multiple times, you realize that they've actually got some pretty great qualities and you're glad to have made their acquaintance.

FOREIGN FUNCTION INTERFACE (FFI)

Another of Rust's features that particularly appealed to me is Rust's support for efficient C bindings (calling C code from Rust incurs no additional overhead). Efficient C bindings support incremental rewriting of software, allowing programmers to leverage the large quantities of C code that are not going away anytime soon. External functions fall beyond the protections of Rust and thus are always assumed to be unsafe (it is important to note that there are many behaviors, such as deadlocks and integer overflows, that are undesirable, but not explicitly unsafe in the Rust sense). In Rust, unsafe actions must be placed inside an `unsafe` block. Inside the `unsafe` block, Rust's wilder crazier cousin "Unsafe Rust" rules. "Unsafe Rust" is allowed to break limited sets of Rust's normal rules, the most important being that it is allowed to call external functions.

In practice, calling C functions from Rust isn't always quite so straightforward as tutorials make out. Consider calling the function `dtrace_open()` (from `libdtrace`). The C prototype for `dtrace_open()` is shown below:

```
dtrace_hdl_t *
dtrace_open(int version, int flags, int *errp)
{
      ….
}
```

To call `dtrace_open()` from Rust, we first specify the `dtrace_open()`'s signature in an extern block (`extern "C"` indicates the call uses the platform's C ABI). We can then call that function directly from an unsafe block, as shown:

```
extern crate libc;

...

extern "C" {
    fn dtrace_open(arg1: ::std::os::raw::c_int,
         arg2: ::std::os::raw::c_int,
         arg3: *mut ::std::os::raw::c_int) -> *mut dtrace_hdl_t;
}

fn main() {
    let dtrace_version = 3;
    let flags = 0;
    Let mut err = libc::c_int = 0;
    let handle = unsafe {
         dtrace_open(dtrace_version , flags, &mut err)
    };
}
```

But there is one significant problem: where is the type `dtrace_hdl_t` defined? While `dtrace_hdl_t` can be specified by hand, it contains many, many fields, which in turn use yet more new types that must be defined. Specifying all this by hand would be extremely tedious and error prone. Fortunately, there is a solution. C bindings can be generated automatically using Rust's bindgen crate (`cargo install bindgen`). Unfortunately, bindgen is not a very mature tool. And, as a result, manually tweaking of its outputs is often required (usually adding or removing mutability). With SWIG support for Rust not looking imminent, better native tooling for generating Rust bindings is desperately needed.

PACKAGE MANAGEMENT

The final, and in many ways most important feature that attracted me to Rust was its support for modern application package management. Rust provides a flexible system of crates and modules for organizing and partitioning software and managing visibility. Rust crates are equivalent to a library or package in other languages, and Rust modules partition the code within the crate. A Rust program typically consists of a single executable crate which optionally has dependencies on one or more library crates. Reusable, community-developed library crates are hosted at crates.io, the central package repository for `cargo`, Rust's package management tool (crates.io is broadly equivalent to Python's PyPI). Rust's `cargo` tool fetches project build dependencies from crates.io and manages building of the software. Yeah, I know, does the world really need yet another mechanism for packaging software, resolving dependencies, and building software? Well perhaps not, but `cargo` actually works really well (though for those with experience with Maven, the bar hasn't been set that high).

GETTING STARTED ON FREEBSD

Rust's platform support is divided into three tiers, each providing a different set of guarantees. FreeBSD for x86_64 is currently a Tier 2 platform. That is, it is guaranteed to build, but not to actually work. Despite the lack of a guarantee, in practice, things generally seem to work pretty well. Tier 2 platforms provide official releases of the Rust compiler rustc and standard library std (`pkg install rust`), and package manager cargo (`pkg install cargo`). FreeBSD's binary Rust package is currently (at the time of writing) at v1.12 with v1.13 being the latest stable release. Once installed, Rust can be updated to the latest version by executing the rustup script:

```
curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

32-bit FreeBSD sits in Rust's lowly third tier where, without guarantees about either building or working, things are pretty unstable. For example, Rust 1.13 recently shipped in spite of a serious code generation bug on ARM platforms using hardware floating point. Here be dragons, so beware!

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.

We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at http://www.emc.com/careers or send direct inquiries to karl.augustine@dell.com.

EMC² ISILON

WHERE ARE WE NOW?

Rust started life in 2009 as a personal project by Mozilla employee Graydon Hoare. In subsequent years, Rust has transitioned to a Mozilla-sponsored community project with over 1,200 contributors. Since the 1.0 release, delivered in June 2015, Rust has been used in a number of real-world deployments. June 2016 saw another major milestone on the road to maturity, with Mozilla shipping Rust code for the Servo rendering engine in Firefox 48.

So people are using Rust, but does it really deliver on its vision of providing a safe alternative to C++? I think the answer is pretty much yes, though the differences aren't all that huge. For example, in C++, a `unique_ptr` owns and manages an object and disposes of that object when the `unique_ptr` goes out of scope. Furthermore, ownership can be transferred using `std::move`; and as a bonus, there is type inference using the `auto` keyword. But in spite of these similarities, smart pointers don't give everything that Rust's ownership system does. In the example below [3], accessing `orig` after the `move` results in a segmentation fault at runtime—a morally equivalent example in Rust would fail to compile. Failing early is a good thing. That a careful and skilled C++ programmer wouldn't make such mistakes is somewhat of a circular argument, because if such mistakes weren't widespread, languages attempting to prevent them wouldn't exist in the first place. C++ also lacks a module system and has a number of pretty ropey features like header files and textual inclusion. These are all wins for Rust.

```
#include <iostream>
#include <memory>

using namespace std;

int main ()
{
    unique_ptr<int> orig(new int(5));

    cout << *orig << endl;
    auto stolen = move(orig);
    cout << *orig << endl;
}
```

How does Rust compare with C++ on performance? Well, control studies comparing the performance of idiomatic C++ and Rust are hard to find. A comparison between Firefox's Servo and Gecko rendering engines (written in Rust and C++, respectively) reported that the Servo engine was in the order of twice as fast [1]. While these figures should be taken with a pinch of salt, the consensus opinion is that Rust is at least comparable in terms of performance to C++. One of the reasons for this is that Rust features like genuine immutability allow optimizations that can't be made in C++. And Rust's semantics bring significant potential for further optimizations.

Despite the advances made in deploying Rust in production environments, problems remain. The Rust ABI is unstable; and as with the Glasgow Haskell compiler, a stable ABI may never happen, almost certainly not anytime soon. This problem most impacts Rust native, shared libraries, as without a stable ABI, they are incompatible across major version changes. ABI instability isn't a show stopper. So is there a technical barrier to upstreaming Rust code to FreeBSD, for instance? In my opinion, I don't think so, but I'd be interested to hear others' opinions on the both technical and political challenges of doing so.

I like Rust. It's fun. And isn't that what really makes us come into work in the morning? ●

GRAEME JENKINSON is Senior Research Associate in the University of Cambridge's Computer Laboratory, leading development of distributed tracing for the Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS) project. Prior to working on CADETS, he has 13 years' experience working in the defense and automotive industries.

REFERENCES

[1] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. "Engineering the servo web browser engine using Rust," *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 81–89). ACM. (May 2016)

[2] L. Hatton. *Safer C*. 1st ed. London: McGraw-Hill. (1995)

[3] S. Klabnik. *Unique Pointer Problems*. Steve Klabnik's Home Page. Available at: http://www.steveklabnik.com/uniq_ptr_problem/. (Accessed November 29, 2016)

[4] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. "Into the depths of C: elaborating the de facto standards," *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 1–15). ACM. (June 2016)

[5] *The Rust Programming Language*. "Getting Started." Available at: https://doc.rust-lang.org/book/getting-started.html. (Accessed November 29, 2016)

SEE TEXT ONLY

Swift Programming Language

·Private·Fileprivate·

·Internal·Public·Open·

by Steve Wills

Swift is a new general purpose programming language from Apple that was announced at Apple's annual WWDC event in 2014 and released as open source in 2015 under version 2.0 of the Apache License. It is designed to replace Objective C, but to be more concise and safer, and has been. Swift is primarily targeted at iOS application development, but it is a complete general purpose systems programming language and can be used for many tasks. It is under rapid development and is portable to many operating systems. Swift fully supports Unicode. It's easy to learn, fun to use, and fast to run. Swift supports both imperative and object-oriented programming, as well as generics, extensions, try/throw/catch error handling, dynamic dispatch, extensible programming, and late binding.

Safety

Memory is managed automatically, so there's no need to manually allocate or free it, which avoids errors. As part of its design goal of making programming safer, Swift largely avoids exposing pointers to the developer, though it is possible to work with them where needed. It requires variables to be declared before use, avoiding issues with implicit declaration sometimes seen in scripting languages. Types may be inferred for the sake of conciseness or declared where needed for safety and clarity. Data types are strictly enforced for safety, but types can be cast easily for flexibility.

Five different access controls are supported for symbols (variables, functions, classes, etc.):

- Private - accessible only in the immediate scope
- Fileprivate - any code within the same file may access
- Internal - any code within the same module may access
- Public - accessible from any module
- Open - may be subclassed outside of the module (classes and methods only)

These access controls apply regardless of inheritance and allow explicit control of where code and data are used, avoiding surprises and difficult troubleshooting.

How to Build It on FreeBSD

First, ensure you have FreeBSD 11 and the latest FreeBSD ports tree. Then:

```
cd /usr/ports/lang/swift ; make install
```

Swift uses a custom version of llvm, clang, lldb, cmark, and llbuild, so the build will take some time. After this, you should be able to run the "swift" command and get an interactive prompt. You can also save Swift files and call the Swift interpreter on them.

What's Missing?

Swift can be interpreted or compiled, but currently only the interpreter works on FreeBSD. As of this writing, the lang/swift port is version 2.2.1, while the current release of Swift is 3.0.1. Work is underway to update the port and enable the compiler. Swift tutorials are often focused around writing iOS apps, but those require libraries that are part of iOS, which at this time are only available on iOS. Therefore, only the core Swift programming language is available on FreeBSD.

Hello, World!

As with any programming language, the first program we write in Swift is one that prints "Hello, World". In Swift, it's as simple as:

```
print("Hello, World!")
```

When entered into the interpreter, it looks like this:

```
(swift) print("Hello, World!")
Hello, World!
```

Swift doesn't require semicolons at ends of lines, but does allow them:

```
(swift) print("Hello"); print("World!");
Hello
World!
```

This helps keep the code easy to read, while also allowing developers to be as concise as they want.

Variables and Constants

Variables must be declared before they are used:

```
(swift) var message = "Hello, World!"
// message : String = "Hello, World!"
(swift) print(message)
Hello, World!
```

Again, the goal of conciseness is served by allowing the developer to skip declaring the variable type since Swift has automatically determined that our message is a string type. Of course, where needed or desired, variable types may be declared:

```
(swift) var message2: String = "Hello"
// message2 : String = "Hello"
```

Whoops, we forgot part of our message. Let's add it:

```
(swift) // add rest of message
(swift) message2+=", World!"
(swift) print(message2)
Hello, World!
```

Once again, conciseness is served by allowing simple string concatenation.
We can also specify constants:

```
(swift) let message3 = "Hello"
// message3 : String = "Hello"
```

Which can't be modified:

```
    (swift) message3+=", World!"
    <REPL Input>:1:9: error: left side of mutating operator isn't mutable:'message3'
is a 'let' constant
    message3+=", World!"
    ~~~~~~~~^
    <REPL Input>:1:1: note: change 'let' to 'var' to make it mutable
    let message3 = "Hello"
    ^~~
    var
```

This serves the goal of safety by allowing the programmer to specify some values as read-only where necessary. The error message makes it easy to see where to change things if we want to make a variable read/write.

Types

Swift provides all C and Objective-C types, including Int, Double, Float, Bool, and string for textual data.

```
(swift) var four: Float = 4
// four : Float = 4.0
(swift) var five: Double = 5
// five : Double = 5.0
(swift) let six: Float = 6
// six : Float = 6.0
```

Once again, safety is served by enforcing variable type:

```
     (swift) let result = four + five
     <REPL Input>:1:19: error: binary operator '+' cannot be applied to operands of
type 'Float' and 'Double'
     let result = four + five
                  ~~~~ ^ ~~~~
     <REPL Input>:1:19: note: overloads for '+' exist with these partially matching
parameter lists: (Float, Float), (Double, Double)
     let result = four + five
                       ^
     (swift) let result = four + six
     // result : Float = 10.0
```

But ease and flexibility are served by allowing them to be cast easily:

```
(swift) let result2 = four + Float(five)
// result2 : Float = 9.0
```

Swift helps us read numbers more easily:

```
(swift) let million = 1_000_000
// million : Int = 1000000
(swift) print(million)
1000000
```

And once more, conciseness is served since Swift allows us to assign multiple variables at once:

```
(swift) var (foo, bar, baz) = (10, 100, 1000)
// (foo, bar, baz) : (Int, Int, Int) = (10, 100, 1000)
```

In addition, Swift provides Array, Set, Dictionary, Ranges, and Tuples:

```
(swift) let myarray: Array<String> = ["first", "second", "third"]
// myarray : Array<String> = ["first", "second", "third"]
(swift) let secondarray = [1: "Alice", 2: "Bob", 3: "Chuck"]
// secondarray : [Int : String] = [2: "Bob", 3: "Chuck", 1: "Alice"]
(swift) let People: Dictionary<Int, String> = [1: "Alice", 2: "Bob", 3: "Chuck"]
// People : Dictionary<Int, String> = [2: "Bob", 3: "Chuck", 1: "Alice"]
(swift) let range = 0...5
// range : Range<Int> = Range(0..<6)
(swift) let range2 = 0..<10
// range2 : Range<Int> = Range(0..<10)
```

Swift also introduces the innovative concept of Optionals, variables where values may or may not be present:

```
(swift) var daytime: Bool? = true
// daytime : Bool? = Optional(true)
(swift) var mayContainNumber = "404"
// mayContainNumber : String = "404"
(swift) var actualNumber = Int(mayContainNumber)
// actualNumber : Int? = Optional(404)
```

Swift implied that our actualNumber may or may not contain a number, rather than forcing us to specify it. We use the "!" operator to get the value:

```
(swift) print("actualNumber has an integer value of \(actualNumber!).")
actualNumber has an integer value of 404.
```

However, we must be safe and check that the value exists before using it:

```
(swift) actualNumber = nil
(swift) print("actualNumber has an integer value of \(actualNumber!).")
fatal error: unexpectedly found nil while unwrapping an Optional value
```

So, we must write:

```
(swift) if actualNumber != nil {
            print("actualNumber has an integer value of \(actualNumber!).")
        }
(swift) actualNumber = Int(mayContainNumber)
(swift) if actualNumber != nil {
            print("actualNumber has an integer value of \(actualNumber!).")
        }
actualNumber has an integer value of 404.
```

Type Safety

Unlike C, the assignment operator does not return a value, so this produces a nice error message and once again keeps us safe from easy typos:

```
    (swift) if foo = bar {
                print("fail")
            }
```

CODE CONTINUES ON PAGE 14

CONTINUED FROM PAGE 13

```
<REPL Input>:1:8: error: use of '=' in a boolean context, did you mean '=='?
if foo = bar {
    ~~~ ^ ~~~
          ==
```

And also, helpfully suggests the necessary change.
Similarly, integers cannot be used as booleans:

```
  (swift) let i = 1
  // i : Int = 1
  (swift) if i {
            print("foo")
          }
<REPL Input>:1:4: error: type 'Int' does not conform to protocol 'BooleanType'
if i {
   ^
```

Along with the requirement that variables must be declared before use, these requirements eliminate several classes of common mistakes.

Control Flow

Swift has the usual control flow mechanisms, "if" and "switch" for conditionals, and "for" and "while" loops:

```
var value = 10
// Note {} are required for "if" and "while", even with just one statement
if value > 0 {
  print("true")
}

// Note lack of break statements, cases do not fall through by default, but
// can with "fallthrough"

var weekday = "Thursday"
switch weekday {
case "Monday":
  print("They call it stormy Monday")
case "Friday":
  print("TGIF")
default:
  print("Is it Friday yet?")
}

// Note use of Range and "in"
for value in 1...10 {
  print("\(value) * = \(value * 9)")
}

while value > 0 {
  print("Not yet")
  value -= 1
}
```

Functions

Swift uses named arguments to functions, which makes the code easier to read and maintain, though in Swift 2.x the first name must be left off. Functions may or may not return a value. Take this example code (on next page):

```
// function which greets user
// returns the greeting
func hello(user: String) -> String {
  let result = "Hello, " + user + "!" // result not modified, so constant
  return result
}

/* function which greets user a second time
   (comments may be of either style) */
func helloAgain(user: String) -> String {
  return "Hello again, " + user + "!"
}

// function which takes additional bool arg
func hello(user: String, seen: Bool) -> String {
  if seen {
    return helloAgain(user)
  } else {
    return hello(user)
  }
}

// greet multiple users, doesn't return a value
func helloAll(users: [String]) {
  // Ensure we were given users, otherwise next line would error
  guard users.count > 0 else { return }
  users.forEach { user in
   print(hello(user, seen: false))
   }
}

let users = ["Alice", "Bob", "Chuck"]
var seenUsers = ["Alice": true, "Bob": false]
seenUsers["Chuck"] = false
var nousers: Array<String> = Array()
helloAll(users)
print(hello("Alice", seen: true))
helloAll(nousers)
```

Save it to a file called multihello.swift and run it:

```
$ swift multihello.swift
Hello, Alice!
Hello, Bob!
Hello, Chuck!
Hello again, Alice!
```

Variadic functions are supported via "...":

```
func sum(values: Int...) -> Int {
    var sum = 0
    for value in values {
         sum += value
    }
    return sum
}
sum(10, 20, 30)
```

Function nesting is supported:

```
func returnValue() -> Int {
   var i = 1
   func add() {
       i += 1
   }
   add()
   add()
   add()
   return i
}
returnValue()

(Returns 4)
```

And since functions are a first class type, they may be returned from other functions:

```
func returnAdder() -> ((Int) -> Int) {
     func adder(number: Int) -> Int {
          return 1 + number
     }
     return adder
}
var adder = returnAdder()
adder(10)
```

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations,
and offers friendly, knowledgeable support staff.
Starting at just $20/mo you are granted access to the latest
FreeBSD, full Root Access, and Private Cloud options.

www.rootbsd.net

Classes

Swift is also object oriented. Classes can be declared like so:

```
  class Building {
     var numberOfFloors = 0
     init(numberOfFloors: Int) {
        self.numberOfFloors = numberOfFloors
     }
     func getDescription() -> String {
        return "A building with \(numberOfFloors) floors."
     }
}

class PaintedHouse: Building {
     var color: String

     init(floors: Int, color: String) {
        self.color = color
        super.init(numberOfFloors: floors)
     }
     override func getDescription() -> String {
         return "A \(color) building with \(numberOfFloors) floors."
     }
}

var house = Building(numberOfFloors: 2)
var houseDescription = house.getDescription()

let myHouse = PaintedHouse(floors: 3, color: "blue")
print(myHouse.getDescription())
```

Finale

There's much more to Swift, and I encourage everyone to give it a try. There are many good online resources on learning Swift:

https://swift.org/
https://developer.apple.com/swift/
https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.html

There are even several that allow you to run code online via web browser:

https://www.weheartswift.com/swift-sandbox/
https://swiftlang.ng.bluemix.net/#/repl

STEVE WILLS is a husband and father living in North Carolina. He is a FreeBSD ports committer with a focus on Ruby and other programming languages.

SEE TEXT ONLY

THE LLD LINKER IN FREEBSD

A LINKER IS A PROGRAM THAT TAKES ONE OR MORE OBJECT FILES GENERATED BY A COMPILER OR ASSEMBLER AND COMBINES THEM INTO ONE EXECUTABLE PROGRAM OR LIBRARY. IT IS PART OF AN OPERATING SYSTEM'S TOOLCHAIN, THE SET OF PROGRAMS USED TO BUILD, DEBUG, AND TEST SOFTWARE.

by Ed Maste

From the beginning, FreeBSD has used the GNU binutils linker. It is also known as the "BFD" linker, after the Binary File Descriptor library upon which it is built. FreeBSD developers kept the copy of the linker in the FreeBSD tree up-to-date, importing new versions as they were released by the GNU project.

Then, in the mid-2000s, the Free Software Foundation (FSF) changed the license to version 3 of the GNU General Public License (GPLv3). It added new restrictions that some FreeBSD developers and users found unpalatable. Since that time, developers in the FreeBSD Project have continued to apply minor updates and bug fixes, but the linker has not been updated beyond version 2.17.50.

The need for a new linker for FreeBSD has been apparent for some time, and a viable candidate emerged recently: LLD, the linker in the LLVM family of projects. LLD is intended to be a high-speed linker with support for multiple object file formats. It supports the ELF format

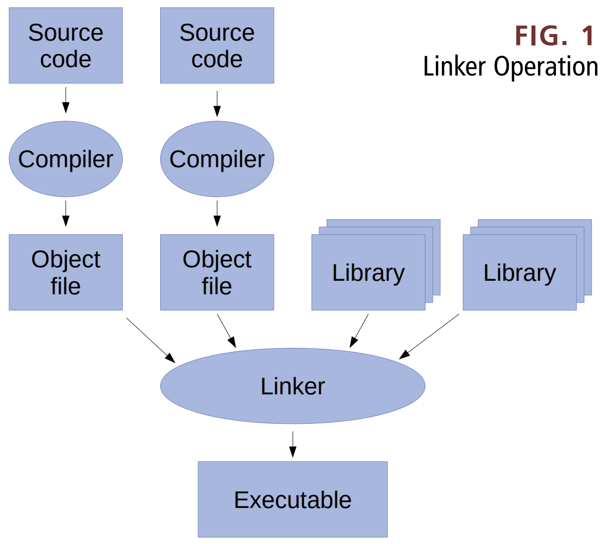


FIG. 1 Linker Operation

used by Unix-like operating systems, Windows' COFF format, and Darwin/OS X's Mach-O.

Where possible, LLD maintains command-line and functional compatibility with existing linkers such as GNU ld, but LLD's authors are not constrained by strict compatibility where it hampers performance or desired functionality. As with other LLVM projects, LLD is released under a permissive Free Software license.

LLD can deliver for FreeBSD a modern, maintainable, and high-performance linker for the base system. It will allow us to support the new CPU architectures in FreeBSD using an in-tree toolchain, and will enable new performance optimizations like Link-Time Optimization (LTO).

HISTORY

LLD was added to the upstream LLVM source repository at the end of 2011, with a design based largely on linking requirements of the Mach-O format used by Apple and the atom model it implies. Atoms are the smallest indivisible chunks of code or data, and are a rather generic representation of an object file. Along with Mach-O support, LLD initially included an atom-based ELF and COFF linking implementation.

Linking the Mach-O format needs the flexibility afforded by the atom model, but it's an unnecessary complication for the ELF and COFF formats. For these formats, the section is the smallest usable unit.

In May 2015, Rui Ueyama of Google started working on a new section-based COFF linker, and two months later in July, Michael J. Spencer committed a new ELF linker implementation based on the section-based COFF support. The new COFF implementation was enabled by default in August, and likewise for ELF in November.

By the end of 2015, LLD could form part of a self-hosting FreeBSD amd64 toolchain: that is, it was possible to build Clang/LLVM and LLD using Clang/LLVM and LLD. Many developers improved LLD throughout 2016, with notable FreeBSD-related contributions from Rui Ueyama, Rafael Espindola, George Rimar, and Davide Italiano.

Over the past year, I have been experimenting with building the FreeBSD base system with LLD. Although LLD could self-host at the end of 2015, it lacked support for many features required by FreeBSD. I created a bug report in LLVM's bug tracker as a "meta bug" to keep track of all of the issues preventing the use of LLD as a FreeBSD system linker. Over time it grew to 63 individual issues, of which seven remain open.

Issues included lack of relocatable output, fine-grained control over library search paths, arithmetic expressions in linker scripts, comprehensive versioned symbol support, and miscellaneous command-line options. I had to disable the build of FreeBSD's boot loaders, 32-bit compatibility libraries, tests, rescue binaries, and the GDB debugger in order to link a subset of userland binaries, although many did not run. Linking the kernel was not possible, as it relies extensively on the use of a linker script.

By March 2016, with a few temporary workarounds in FreeBSD and LLD, I was able to build a usable subset of the FreeBSD userland. Symbol versioning and linker script expressions were still unsupported, and the kernel could still not be built as a result. However, the rapid progress of LLD's development convinced me it was on track to become a viable system linker.

Symbol versioning and linker script expression evaluation arrived a few months later. By August, relocatable output and a somewhat esoteric command line option used in building boot loader components were the significant features outstanding. Now LLD is usable for nearly all aspects of the amd64 FreeBSD base-system userland and kernel build; the boot loaders are the only FreeBSD components that still do not build with LLD.

DESIGN

LLD's design goals include speed, simplicity, and extensibility. LLD attempts to be fast by doing less work, and when it is necessary to do something, doing it only once. Traditional UNIX linkers visit objects sequentially, including objects specified directly on the link command line and those included in an archive. The linker builds a list of undefined symbols, and links those objects that satisfy the requirements for those symbols. The new objects may in turn introduce new undefined symbols, and objects may need to be visited multiple times to resolve all undefined symbols.

LLD, instead, performs only one pass over the specified object files and archives, keeping track of both defined and undefined symbols found in each object it visits. Symbols are resolved without having to revisit objects or libraries again. This results in slightly different linking semantics when compared to traditional linkers, but well-written software should see no difference. It is possible to craft a scenario that works with a conventional linker and fails with LLD, but that is expected to be unlikely in practice. No broken cases have yet been found while testing a variety of third-party software with LLD as the linker.

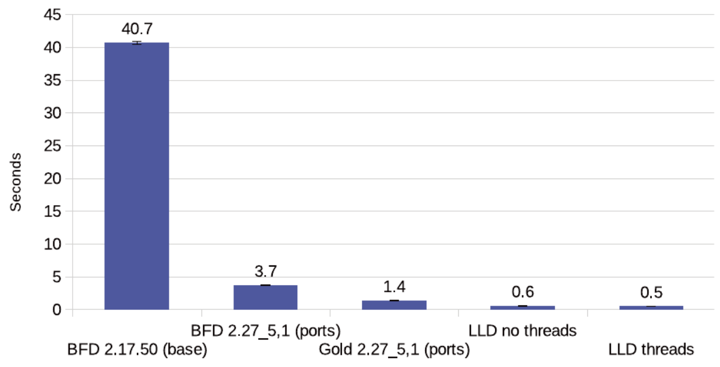
The ELF and COFF linkers in LLD share the same general design, but do not share code. They provide the same command line user interface as the native linkers for each file format: GNU ld for ELF, and Microsoft's linker for COFF. This avoids the complexity and runtime cost of an abstraction layer. As a result of this approach, the ELF linker is only about 13,000 lines of code. This number is much smaller than other linkers (GNU ld and GNU gold), even though it isn't directly comparable, as all three rely in different amounts on support libraries.

As with other components in the LLVM family, the linkers in LLD are implemented as libraries with a lightweight command line driver. This allows it to be easily embedded in other projects.

PERFORMANCE

One of LLD's primary design goals is to be a high-performance linker. Small programs should link quickly, and linking should not become exponentially larger as program size increases.

FIG. 2 Real Link Time, Clang Release Build

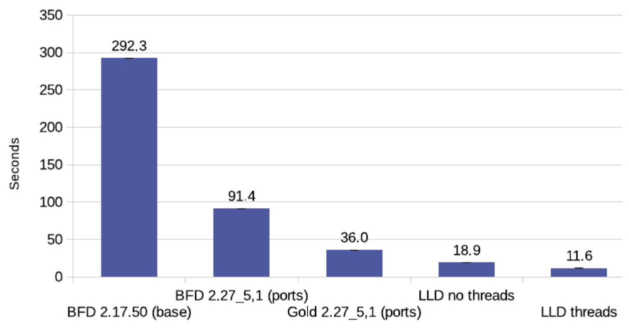


The graph in Figure 2 compares the real (wall-clock) time taken to link a release build of the Clang compiler using several different linkers. The linker inputs consist of 158 object (.o), static library (.a), and shared library (.so) files, totaling 91 megabytes in size.

The experiment was performed on my development desktop, which has a quad-core (8 thread) Intel i7-3770 CPU and 32GB of memory. I compared BFD ld 2.17.50 (the current FreeBSD base system linker), BFD and Gold 2.27_5,1 from the FreeBSD ports tree, and LLD built from the LLVM source repository. By default, the Gold linker in the FreeBSD ports collection is built without threading support, and Gold's threaded mode was not investigated further.

Figure 3 compares the same linkers for a debug build of Clang. The linker processes the same number of files in this case, but the addition of

FIG. 3 Real Link Time, Clang Debug Build



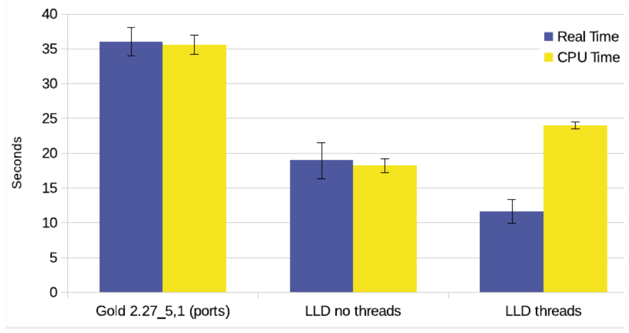
debug data makes them substantially larger and requires the linker to do much more work.

LLD achieves this high performance in several ways that can be briefly enumerated as: avoid doing work where possible, perform expensive but necessary operations only once, and use threads to perform operations in parallel.

Several tasks in LLD can be performed in parallel, including uncompressing input sections, splitting mergeable sections, merging common strings, and identical code folding. Figure 4 demonstrates the effect of LLD's use of threading. The link completed 63% faster, but consumed about one-third more CPU time.

This trade-off is worthwhile in a typical devel-

FIG. 4 Real & CPU Link Time, Clang Debug Build



oper's edit-compile-test cycle. The final linking step requires all of the individual compiler invocations to be complete, and there is likely no other work the computer could do. When linking software on a shared resource (for example, building the FreeBSD package sets), disabling threads may be a better choice.

ARCHITECTURAL SUPPORT

LLD includes at least some support for almost all the CPU architectures relevant to FreeBSD. As described earlier, amd64 (or x86-64) is well supported, and LLD built from the development repository is capable of linking a working FreeBSD/amd64 kernel and userland, except for the boot loaders.

32-bit x86 (i386) and AArch64 (arm64) support is also quite mature, but not yet well tested in FreeBSD. LLD can self-host on arm64 and link a working FreeBSD/arm64 kernel, with small workarounds for outstanding issues.

32-bit ARM, 32-bit and 64-bit MIPS, and 32-bit and 64-bit PowerPC are supported by LLD, and are capable of at least linking trivial applications, but are not yet viable as a FreeBSD system linker.

RISC-V support is being planned, but has not yet started. It seems that sparc64 is the only FreeBSD CPU architecture that is unlikely to be supported by LLD.

LINKER OPTIMIZATIONS

One large benefit LLD will bring to FreeBSD is base system support for whole-program Link-Time Optimization (LTO). LTO refers to optimization that is performed across modules at link time.

A common optimization is to eliminate unused code paths. With conventional linking, this can be applied only within a single source file. With LTO, the compiler and linker work together and can eliminate code paths that can only be determined as unused when examining the whole program.

For LTO the compiler emits an LLVM bitcode file instead of native machine instructions in an ELF object file. The linker processes these bitcode files similarly to regular object files and allows both types to be used together.

Another optimization can be performed by the linker. Large C++ applications often contain functions that compile to machine instructions identical to another function. Identical Code Folding (ICF) is an optimization that identifies read-only sections that happen to have the same content. For a sample of large binaries, ICF reduces the size by 5% to 8%.

NEXT STEPS

Clang/LLVM 3.9 was recently imported into FreeBSD's development branch, and that work included LLD 3.9. LLD is now installed as /usr/bin/ld.lld on amd64 and arm64 for experimentation and testing. On arm64, it is also now installed as ld, as the BFD linker version 2.17.50 does not include arm64 support.

Linking the boot loaders with LLD must be addressed next. LLD developers are actively working on issues in LLD that prevent this, and some changes may also be required in FreeBSD. Once this is complete, a newer snapshot of LLD will be imported into FreeBSD and made available with a build-time configuration setting (for example, WITH_LLD_AS_LD=yes). The ports tree will then be extensively tested using the in-tree LLD as the system linker.

The investigation and iterative bug fixing approach will need to be undertaken on all of FreeBSD's supported CPU architecture. Experimentation with linker optimizations will then proceed for both the FreeBSD base system and ports collection. ●

ED MASTE manages project development for the FreeBSD Foundation and works in an engineering support role with the University of Cambridge Computer Laboratory. He is also a member of the elected FreeBSD Core Team. Aside from FreeBSD and LLVM projects, he is a contributor to several other open-source projects. He lives in Kitchener, Canada, with his wife, Anna, and sons, Pieter and Daniel.

SEE TEXT ONLY

TEACH BSD

By Benedict Reuschling and George Neville-Neil

The second week of August 2016, George Neville-Neil visited the Computer Science department at the University of Applied Sciences, Darmstadt, Germany, where I work. George has worked with Robert Watson at the University of Cambridge developing a master-level course over the last several years.

● The course uses DTrace to analyze and explore a live FreeBSD operating system. I had arranged for George to come to my university to teach an undergraduate version of the course to a group of interested students in the inter-semester holidays. This endeavour had many goals: we were curious how the course material, available online at teachbsd.org, would work for undergraduate students, especially those who had not taken operating systems courses before. Another goal was to adopt the course to contain less paper writing, which is emphasized in the masters version of the course. while still focusing on practical content. Lastly, we wanted to see how well the German students would understand a course taught in English and whether they would be willing to ask questions and participate in class. We were pleasantly surprised to find that English was not a barrier for the students and they soon started asking questions.

The course was divided into morning and afternoon sessions with a break for lunch. The morning sessions were typical lectures where George explained how different kernel subsystems work, giving both the theoretical background necessary to understand the concepts as well as an overview of the implementation in a modern operating system, FreeBSD. After lunch, the students were given lab assignments to solve using DTrace on virtual machines. The labs covered the topics of what had been taught during the morning lectures. Due to the fact that these guest lectures were scheduled over the semester holidays, we had only a handful of students participating. The small class size created a situation where we were able to give more individual help to students, throughout the week, which is usually not possible with a larger group.

Monday started with an overview of the course content, including a bit of computer history. The introduction and overview sections introduce students to the origins of operating system development, its evolution, and reasons for some of current OS implementations. Many parts of modern operating systems come from the interaction between hardware and software advances, with one influencing the other. One clear example is that modern RISC processors were influenced by the rise of UNIX and the C language; this type of tension motivates many of the lessons we presented to the students. The goal of the lab on Monday was to get the students set up with their virtual machines and to familiarize them with DTrace as a tool, letting them get their hands dirty and take their first, basic, traces.

Tuesday morning started with a short quiz to determine the students' current knowledge level and to gauge how well Monday's topics were understood. We then continued by explaining how processes work, what virtual memory is, and how processes and threads are scheduled in the operating system. A lot of attention was paid to the topic of locking, and what challenges it poses, including deadlocks and lock order reversals. Without a clear understanding of concurrency and locking, students cannot really appreciate many of the subtleties of modern hardware and software, whether in an operating system or in a multi-threaded user program. The lab covered tracing processes and their various states, getting the students to explain the UNIX process life cycle. Feedback from the second day of the course was turned into upated slides. More than once, the night after a class was spent revising or creating whole new sets of slides to illustrate concepts that students struggled to understand during the day.

With the basics of processes, virtual memory, and concurrency having been covered, we then moved on to communication in the operating system. Starting with basic Interprocess Communication, we explained signals and how they are used by programs and the operating system to communicate. From basic IPC we moved on to sockets and network communication. Networking topics, including DNS, UDP, and TCP, were covered with an emphasis on how the operating system implements TCP—connection setup, data transfer, sliding windows, and connection tear down. Since all of the students had already completed an undergraduate networking course, they were able to quickly appreciate what the operating system needed to do to implement correct TCP software. Due to this familiarity, the questions the students asked led deeper into the operating system than in previous sections. The communication lab for that day had students trace the TCP connection setup process using a local nginx instance serving a static webpage.
We also had them visualize their derived state machine using the graphviz package, which I taught the students how to use.

The topic of Thursday's lecture was data storage and filesystems. Things like the namecache, virtual filesystem layer (VFS) were explained, as well as reading and writing data. The namecache always requires a good deal of explanation as the concept of caching negative results is one that most undergraduates have not yet been exposed to. In the lab session, the students explored the name cache a bit more and how it interacts with nginx when a certain web page URL is requested. The class day ended a bit early on Thursday afternoon to give students time to study for the exam on the next day.

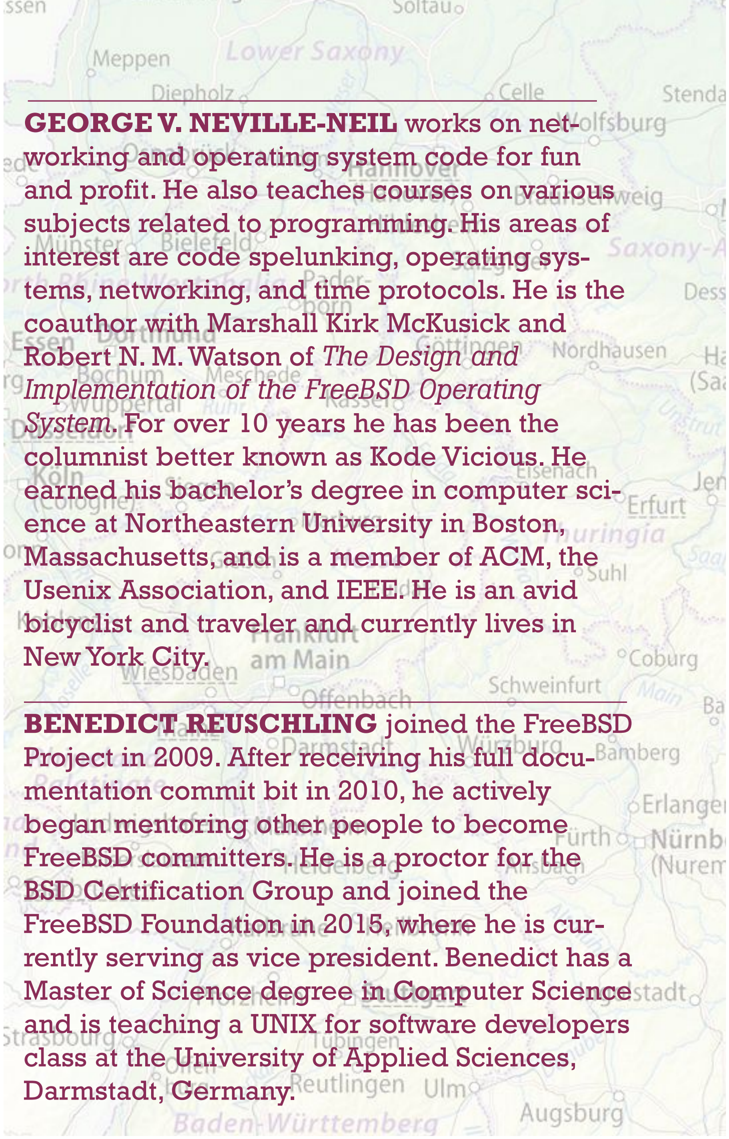
On Friday, we gave the students a whole systems view, combining all the topics discussed during the week. The lecture served not only as a review session before the written exam, but also tied everything together for the students. Specifically, we explored what happens in all the operating system layers when a web server, such as nginx, serves a page. The students were exposed to which worker processes were run, what cache lookups are done, the TCP states that are being created, etc.

The final was a 90-minute exam covering all the topics of the week, with questions focusing on details about the operating system implementation and the writing of a few one-liners for DTrace. Students were allowed to use their laptops throughout the exam in order to test and verify their DTrace scripts before submitting the final versions on their papers. The exam determined the grade for the students in the course. We were very satisfied with the results after grading the exams and we are already discussing how to further enhance the course based on this first test-run.

When George was not teaching (which he enjoyed very much), I was showing him around our campus in Darmstadt, the city, and what good restaurants we have. I also introduced him to some professors who were not on holiday during that week. George not only does great networking on the systems level, but also in person. We had some good discussions, ranging from teaching in computer science to cultural diversity in education and the integration of practitioners' work into the university curriculum.

Some of the more amusing moments of the course related to the inability of all the clocks at the university to keep proper time. Either the batteries were dead, or the timepieces were advancing too slowly or made no sense at all ("nowhere in the world is it 16:00 hours now"). For a timekeeping nerd like George, this proved to be a recurring joke throughout the week.

Overall, it was a very productive week. We gained valuable feedback for the teachbsd course for undergraduates and the students got a lively lecture about operating system internals. I hope to integrate parts of these lectures into my own full-semester course, and George is now planning a return to Darmstadt to teach an updated, and extended, version of the course. ●



GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking, and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.

BENEDICT REUSCHLING joined the FreeBSD Project in 2009. After receiving his full documentation commit bit in 2010, he actively began mentoring other people to become FreeBSD committers. He is a proctor for the BSD Certification Group and joined the FreeBSD Foundation in 2015, where he is currently serving as vice president. Benedict has a Master of Science degree in Computer Science and is teaching a UNIX for software developers class at the University of Applied Sciences, Darmstadt, Germany.

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

SEE TEXT ONLY

INTERACTING / *by George Rosamond*

Starting a BSD User Group

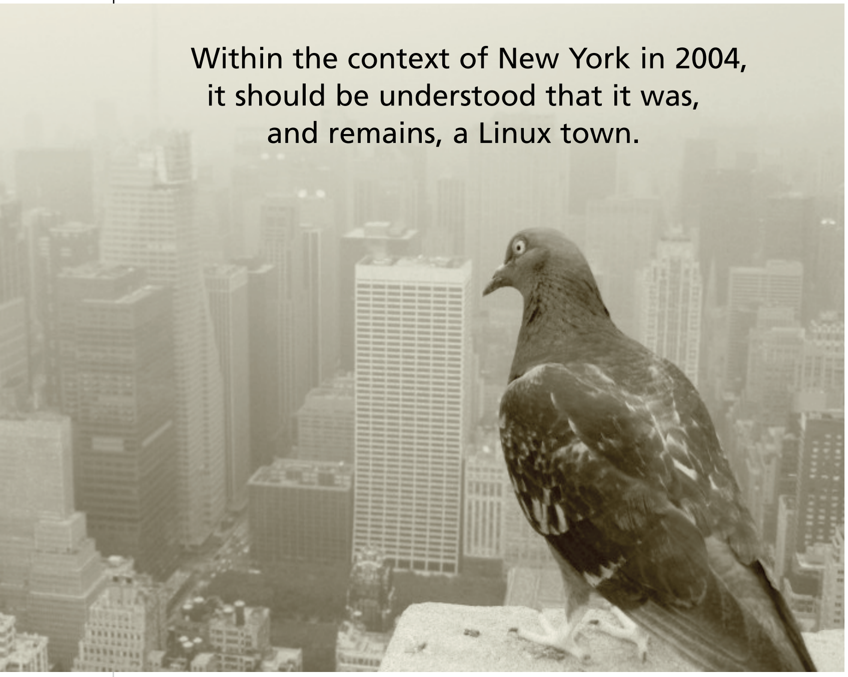
A handful of like-minded BSD users launched the New York City BSD User Group (NYC*BUG) in 2003. NYC*BUG meets monthly and hosts multiple mailing lists, while managing to provide services to the BSD projects and other user groups.

The user group's chronology is filled with the waves and troughs that any technical user group faces and to summarize instructive lessons is difficult. However, I will attempt to review the life of NYC*BUG and present some useful conclusions for others interested in starting a BSD user group (BUG) of their own.

In December 2003, several of us planned to officially launch in January 2004 at what was to be the last Linux Expo in New York City. It was understood that there would be opportunities to engage the broader open-source community at Linux Expo and conduct a birds-of-a-feather session (BoF), culminating in our first meeting in early February.

Some 50 people attended the BoF, and our early meetings were large and engaging. Discussions continued into the early morning hours, and we began to witness how a technical user group could corral people and begin to have an impact.

Within the context of New York in 2004, it should be understood that it was, and remains, a Linux town.



The Unwelcoming Committees

Not everyone in the local community was enthusiastic about our launch. NYC always entertains a myriad of open-source user groups, from broad tents with heavy corporate influence to others that verge on cult status. Yes, cult status.

The local Unix group, in particular, felt we should have run the idea of a BSD group by them before proceeding. We sought neither permission nor forgiveness.

Somewhat more humorously even before the BoF, one post on January 13, 2004, to the NetBSD NYC regional list dismissed us: http://http://mail-index.netbsd.org/regional-nyc/2004/01/13/0003.html.

"i wonder if beer is involved? there's no mention of it in that posting."

"they are just doing what the linux group does... probably same ppl too."

Ultimately, this longtime BSD developer did a meeting for us and became one of our biggest advocates.

Starting with What We Didn't Want

In retrospect, we did dwell in a state of utter confusion. But we were clear on what we didn't want.

A number of us had been around other technical associations and user groups, which all seemed to fall into one of two categories.

First, there were professional associations that lacked any substantial content and seemed more like resume line-items. The respective sector's vendors—such as information security—dominated the presentations and materials, and the passive attendees touched keyboards only for hiring and firing purposes.

Second, technical user groups earned welldeserved reputations as gathering places for the socially awkward, where one or more cliques congregated to exchange the odder ends of technical culture. And like the professional associations, presentations could still be drenched in sales pitches.

We wanted neither scenario. We sought a loose grouping that focused on the BSDs in the production environment.

All about the Asterisk

Another critical element was to be agnostic in relation to BSD projects. While many people in NYC*BUG might strongly favor one BSD over the others, we remain officially neutral. This is not some naive notion. We witness the same occasional mailing list flame wars or nasty interview comments that the rest of the world sees. But fundamentally, we also think the BSDs have more in common than not.

The license, in whichever simple form, remains the clearest overreaching principle. Some may view it as developer- or corporate- or freedom-friendly, but we all are comfortable in having a license that doesn't require a few years of law school to comprehend. If one of the projects imploded tomorrow, we think that the remaining projects would likely provide reasonable new homes for the particular developers and users.

Contrasting the BSD community to other open-source communities also further binds the BSDs together. While criticisms might flow free between the projects within the confines of NYC*BUG, it is all in the spirit of open discussion and without the nastiness characteristic of online flame wars.

Over time, it also became apparent that regular physical meetings undercut the environment for trolling or even making unjustified comments about other BSD projects. The online world allows the worm to imagine itself as a king. But when that worm has to look other worms in the eye, delusions of royalty tend to dissipate.

One meeting in the past year provided an overview of a particular BSD project. There were no other users from that project in the room, and we were curious to see how the discussion would evolve.

The outcome was worth a thousand words. One longtime developer and Unix veteran criticized the project for lack of a focus, but it was done in a manner that more likely pointed to the problems in that person's own BSD project. Another longtime developer humbly spoke about the problems faced by their third BSD project, presenting instruction to the speaker as an open mea culpa. The modesty evoked by face-to-face meetings remains one of NYC*BUG's strengths. All open-source projects are glass houses while online interactions tend to obscure this reality.

Sales People Not Welcome

Another principle that quickly developed was avoiding salespeople, and even implicitly discouraging their attendance. NYC*BUG meetings and mailing lists were no place to preach about some "disruptive" new product, nor a pool for technical recruiters to fish for resumes. Yes, NYC*BUG meetings are free and open to all, but it's the BSDs in production that provide the overarching principle.

There was one particular early trial for this. We asked Apple corporate people in New York for an engineer to speak about Darwin, the BSDs and open source. We remain appreciative and well-aware of the contributions Apple has made to the BSDs, but this particular salesperson insisted such a topic would be nothing more than a 15-minute presentation and implored us to allow the circulation of sales materials. We stuck to our approach, and frankly threatened to end the meeting if the technical presentation became a sales pitch.

The meeting was technical with standing-room-only for over 50 people, with almost no one leaving the room during a two-hour presentation. This confirmed our view. Particularly with so many of us being veterans of the Dot com era, no one was going to voluntarily listen to a sales presentation without an open bar, some expensive giveaways, and maybe a decent steak. We were not looking to occupy a free night each month in our schedules. Rather, we wanted to create an environment where the attendees are active participants and where meeting topics reflected what people were actually doing. It was, and remains, the yardstick by which we measure NYC*BUG's successes and failures.

The Right Relation to Sponsors

Within the context of New York City in 2004, it should be understood that it was, and remains, a Linux town. Until the 2008 crash, the bellwether industry in New York City was finance, often as Linux shops migrated from Solaris. There were a few significant BSD shops in NYC and we set out to engage those shops.

Larger trends matter. The adoption of Linux in the finance industry mattered. But years later, the advent of systemd in the Linux distributions created a new audience for us. Apple's loud support of Unix with the launch of OS X gave us a larger hearing. Noting those larger issues goes a long way in reaching out to broader audiences beyond the usual suspects.

Probably the most fruitful relation built with a BSD shop has been with New York Internet. They had moved from Solaris to FreeBSD years earlier, and were consistently rated one of the best public data centers. While we didn't have the quantity/size of the Linux audience, they were quick to offer us support due to our quality. In one owner's words (paraphrased), "You seem like good people to be around." Since then, NYI has provided us with a full cabinet hosting dozens of projects and has been a consistent sponsor. They have also contributed greatly to the community beyond NYC. FreeBSD's U.S. East Coast infrastructure resides in their New Jersey facility, and both the BSD community and NYI are better off with this relationship. Projects such as the BSD Certification Group, pfSense, to BSD.lv and mandoc, not to mention the many hosted mirrors, all have benefited from the NYI relationship.

"We are asked frequently for advice on starting a user group. The first piece of advice is don't attempt to replicate us."

It's important to connect the notion of "good people to be around" with our attitude to sales presentations, etc. Potential sponsors might want to have free reign on mailing lists and presentations, and even conference attendee lists. But a user group that allows such access probably doesn't attract the types of people that sponsors want to talk to in the first place. Yes, we might make it difficult, but it's well-worth the efforts if our parameters are respected.

Coordinating without Assets

The sponsor question raises an issue of funds and assets. We have conducted five conferences and managed to make a profit on the last four. We maintain the cabinet at NYI. But ultimately, we have no assets, nor do we pursue them. Any profits gained from the conferences are donated back to the BSD projects.

Maintaining a nonprofit entity, or even just a NYC*BUG bank account, demands a higher level of responsibility and accountability than we want. And having such entities means brewing potential conflicts. Being asset-poor means having nothing to fight over. It's safer to be a starfish than a spider, in reference to that pop sociology book. Fork us, copy us, but we are too loose to break. NYC*BUG is more like viscous mud than a sand castle. Cut up a spider into two parts and it dies. Cut up a starfish into two parts, and you have two starfish.

We have maintained a certain informality from the beginning. We strive to allow a fluid membership, which means we can't conduct any meaningful voting. Who is qualified to vote? Mailing list members? What about those who attend one meeting a year? Or those who rarely miss a meeting?

Once the formalisms required by elections are implemented, too many other procedural questions arise. While we explored creating a nonprofit early in our history, we quickly realized that the stakes would be higher, and that being asset poor and as informal as possible was a better route.

Six of us walked into a meeting with a technology-oriented NYC agency that was excited to migrate us into a nonprofit structure. Five of us were on board. But five of us walked out of the meeting against creating a nonprofit, and content to continue without any formal structure.

Outside of a few responsibilities performed by the NYC*BUG admin group, no one is really compelled to do anything. One might attend a meeting when the topic is interesting. Or engage in a mailing list thread when it's relevant to a job or interest. But the regular appearance and reappearance of people became natural, and maintaining this low barrier to entry (and exit) shapes the atmosphere for the better.

But then how can the admin group be a useful body? Admin's purposes are simple. It is not a privilege to be on admin. The central function is to resolve the organizational questions inappropriate for the talk@ mailing list. Discussions might revolve around a colocation issue in the cabinet, or determining upcoming meeting topics. Admin membership evolved from a few of those taking on the roles of coordinating NYC*BUG activities, yet is unelected and there are no formal terms.

For most BUGs, an admin group isn't just unnecessary, it's really a dangerous barrier. If five people are meeting regularly, having two of them decide organizational questions on a separate mailing list is useless. And it will only discourage the other three people from taking an active role. Besides eternally seeking to infuse technical solutions to organizational problems, technical people also tend to overbuild organizational infrastructure as if they are scaling a startup.

Once the admin ball was rolling, we attempted to make it selective in choosing members, yet easy for members to leave admin without any discomfort. It hasn't always been successful, but overall, admin continues to function and provide a decent amount of direction. On occasion, someone from admin will note that some particular individual would be a good addition. We determine their inclusion by consensus, since the chemistry of admin is critical.

One noteworthy incident illustrates admin's role at its best.

Like the broader BSD community, NYC*BUG faces an ugly dearth of diversity, particularly in terms of gender. We are quite conscious of it, and it is a regular discussion. Part of our approach to rectify this is to make it clear that sexism and patronizing attitudes toward women are unacceptable.

In one case, a NYC*BUG talk@ poster used a disgusting and sexist signature in his emails. As this individual had already been a problem on other levels, many people around NYC*BUG had already been sending his posts to /dev/null. But when his signatures appeared, several people not on admin emailed demanding action.

He was promptly removed from the mailing list and the reason was communicated. But it was indicative that other NYC*BUG participants, not on admin, had quickly reacted and believed it was their responsibility to address the situation as well.

Warning: Don't Try This at Home

We are asked frequently for advice on starting a user group. The first piece of advice is don't attempt to replicate us. We are an unusual example. We are in a dynamic, global city filled with technical people and resources. Our meetings frequently include some of the "who's who" of the Unix world. If you're located in a small college town, or in a rural, dispersed area, attempting to photocopy NYC*BUG will lead to endless frustrations.

The overriding lesson is that we started appreciating the context in which we operate. We knew the NYC technical scene. We knew the user group scene. We knew what we didn't want to do. And most importantly, we started modestly with goals we could accomplish reflecting our situation and the resources at hand.

Most of the time when someone wants to initiate a BSD user group, there is one individual driving the effort with the time, energy, and ideas. And that leads to the first problem. There shouldn't be one individual who owns the project. There needs to be at least a handful of those who share a common vision and who can take responsibility for getting the BUG operational, and maintaining it into the future.

On that note, the first step might be initiating a mailing list and publicizing it. Having a launch meeting usually translates into one or two people being viewed as the organizers, and the rest of the attendees passively voting with their involvement, or lack of involvement. A mailing list is a good vehicle for gathering people to take on active involvement from the beginning. Finding common interests and topics is a critical first step.

If you're a handful of people in an hour's radius of, say, Rzeszow in Poland, meeting monthly is difficult. Maybe a quarterly day-long event makes more sense. And why not have everyone engaged in organizing the event? In that case, the mailing list is likely the ideal platform for the group's activities.

Our meetings can range from a dozen to dozens of attendees. For others, four or five is a worthwhile accomplishment. And a meeting on, say, Apache, doesn't have to be presented by a relevant BSD port maintainer. This is a normal mistake. If people in and around the BUG are doing something with the BSDs, there are topics and presenters available. Relying on "names" and topic authorities means the well of speakers dries up quickly. Having actual participants present means keeping the barrier to entry low for everyone. There are a number of speakers at BSDCons whose first presentations were at NYC*BUG meetings. We created the environment that was comfortable and interactive and which ultimately benefits the BSD community beyond NYC.

Come On In. The Water's Great!

Another function that BUGs can play is providing a conveyor belt for individuals to be involved in BSD projects. There is a tendency to view entry into the BSD projects as some virtual meritocracy in which the good rises above the crud, and tomorrow we'll have our next layer of core developers.

Reality is quite different. BUGs can be the place in which meeting developers and other users, face-to-face, can introduce more isolated users into the BSD culture. One would imagine that a lot of eager potential developers find that there's a very valid reason why bash isn't the default shell, or that testing ports on even esoteric architectures does matter.

While NYC*BUG could never take credit for creating developers, a good number of people have gone from being isolated BSD end-users into developers in the environment fostered. Contributing ports, for instance, is a relatively easy path into contributing to the BSDs, and we've attempted to provide introductions that are easy-to-grasp for more users. From there, the slippery slope of spending those few free waking hours quickly evaporates into hacking makefiles and checking dependencies.

Fun, Fun, Fun

Our golden rule remains a phrase from an early USENIX participant: are you having fun?

It is difficult to be motivated to deal with the hassles, the meetings with no-show speakers, etc., if you aren't having fun. Make it fun. If meetings become a chore, maybe make them quarterly. Or just keep the mailing list. Or an IRC channel. Ultimately user groups do not turn into wildly successful IPOs and result in the creation of brilliant new technologies. They are really just social organizations with a purpose.

BUGs Matter

BUGs can provide a useful framework for the community, as an alternative to being grounded in the corporate world or with isolated developers. BUGs can better reflect what actual users are doing, with their needs, desires, and frustrations. Unlike corporate donations, BUGs can provide no-strings-attached contributions. Ultimately, a broader layer of people can support the projects, which in turn also gives a deeper sense of ownership of the projects.

BUGs can provide a useful and grounded picture of production BSD usage in a way surveys can never manage. We spend a lot of time determining meeting topics and speakers, and sometimes we find our choices fall flat. Other times, we find that being creative on meeting topics piques interest no one realized existed. Nothing is more flattering than watching other user groups in NYC, and beyond, replicate our meeting topics and approaches.

NYC still can't really be considered a "BSD City," but the flag we planted certainly fosters that notion.

The BSD community is not traditionally known for its advocacy. But if we had a world with a few dozen BUGs that could maintain mirrors for each BSD project, donate funds and hardware regularly, connect with BSD-using firms in their area, and so on, the BSD community as a whole could be significantly stronger.

NYC*BUG, for instance, created a place where the BSD-curious can dip their toes into the water, and maybe explore how the BSDs could work for them. Such an approach can also work far beyond the confines of New York City. ●

GEORGE ROSAMOND is a founding member of the New York City BSD User Group. By trade a systems administrator, George maintains a small colocation firm for technical clients, in addition to contributing to an array of BSD projects large and small, with a particular focus on privacy-enhancing technologies. For almost two years, his central focus has been the Tor BSD Diversity Project (https://torbsd.github.io/), which looks to extend the usage of the BSDs into the Tor public anonymity network (https://www.torproject.org/).

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with BSD CERTIFICATION

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification **YOU'LL HAVE WHAT IT TAKES!**

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

INTERACTING / by Kristof Provost & Beat Gätzi

EuroBSDcon: Behind the Scenes

EuroBSDcon is an annual technical conference gathering users and developers working on and with the BSD operating systems family and related projects. It is typically held in September/October and hosted in a different European city every year. EuroBSDcon offers an exceptional opportunity to learn about the latest news from the BSD world, witness contemporary deployment case studies, and meet other users and companies using BSD-oriented technologies.

EuroBSDcon is also a boiler plate for ideas, discussions, and information exchanges, which often turn into programming projects.

The conference has always attracted active programmers, administrators, and aspiring students, as well as IT companies at large that found the conference a convenient and quality training option for staff. It usually attracts around 250 attendees and lasts four days: the first two days are tutorial and devsummit days, and the final two days are the conference itself with two to three parallel tracks.

In memory of its first chairman and one of its founders, the EuroBSDcon Foundation offers the Paul Schenkeveld travel grant.

First held in 2001 when the conference started in Brighton, UK, it has since taken place in 12 different countries. The only repeat visits were the conferences in the Netherlands (Amsterdam in 2002 and Maarssen in 2011), the United Kingdom (Brighton in 2001 and Cambridge in 2009), and Germany (Karlsruhe in 2004 and 2010).

Historically the conference organizers changed every year, so in 2010 a nonprofit foundation was created to back the organizers, both financially and organizationally. The EuroBSDcon Foundation is based in the Netherlands.

Nowadays the responsibilities for the conference organization are divided into three parts: the EuroBSDcon Foundation, the organization committee (OC), and the program committee (PC).

The foundation board consists of registered officers with designated roles and a representative of each BSD that is interested in the conference (currently FreeBSD, OpenBSD, and NetBSD) and the head of the OC of the previous, the current, and the next year. Furthermore the foundation has additional members responsible for the website, the registration system, and future members of the board to help prepare them for their roles.

The foundation provides a legal entity to sign contracts and to ensure the financial security of the organizers. It provides a financial buffer so bills can be paid even before the conference registration has opened so things like advances for venues and hotels can be paid. The foundation also hosts the conference website and the registration system, as well as some other basic services like mailing lists.

The foundation also facilitates the transfer of knowledge and experience between organization committees. That's why the heads of the organizing committees of the last and next conferences are also part of the board.

Another big part of the foundation's work is finding sponsors, as about a third of the conference budget is provided by them. The other two thirds of the budget comes from the entrance fees.

The board meets monthly on IRC, and once a year in-person, during the conference itself. Furthermore, some tasks like financial statements and meetings with the accountant require additional in-person meetings of some of the board members.

The organizing committee changes every year and is responsible for the local organization of the conference. The head of the organizing committee is elected by the foundation board.

When selecting a future conference location, the board looks for a city the conference has not been held in before so that frequent attendees have a chance to visit different places, and also so that people who can't afford to travel far are more likely to have a chance to attend. It also tries to alternate between destinations in Western and Eastern Europe for the same reasons.

The organizing committee head ideally serves for two years before the conference is held so that they can observe one full organization cycle before he or she is in charge. The head then forms an organizing committee with other volunteers. The location of the next EuroBSDcon is traditionally announced during the closing session of the current conference.

Immediately after the conference, the organizing committee for the next conference begins the process of selecting a venue and speaker hotel so that the date of the next conference can be announced, at the latest in January.

The OC responsibilities extend beyond the conference itself into things like the social event, the organized dinners, partner program and catering during the conference. They coordinate with the organizers of the devsummits and birds-of-a-feather sessions that take place during the conference. They create a conference logo, design and produce T-shirts, bags, lanyards, speaker gifts, and so on.

The OC also recruits local volunteers to help out at the registration desk, with the partner program, and anywhere else a hand is needed during the conference itself.

The last major player in the organization is the program committee: it is responsible for selecting the talks, tutorials, and keynotes. The PC is formed by the PC head and consists of two members from each of the FreeBSD, OpenBSD, and NetBSD communities. Members of other BSD communities are welcome as well. Once the PC is formed, a call for papers is announced, usually in March. In parallel, the PC searches for interesting keynote speakers. When the call-for-papers period is over, the PC decides which talks and tutorials will be accepted and ensures that the talks are well balanced between the BSDs and also between topics. Once the speakers have been decided, the PC creates the tracks and schedules the talks so that by the end of June, registration can be opened. The PC coordinates the travel and accommodations of the speakers and tutors. During the conference, the PC introduces the speakers and helps them keep to the timing of their talks.

All in all, about two dozen people are involved in organizing the conference. The organizing committee for EuroBSDcon 2017 in Paris has already started work for next year's conference, while the foundation board is still busy with the financial statement of EuroBSDcon 2016.

The EuroBSDcon Foundation is always looking for volunteers to organize a conference in interesting parts of Europe. If you'd like to see a EuroBSDcon in your city, get in touch.

In memory of its first chairman and one of its founders, the EuroBSDcon Foundation offers the Paul Schenkeveld travel grant. Paul was one of the few people, if not the only one, who attended all EuroBSDcon conferences, and as such, the foundation could think of no better way to honor his memory than to enable someone who might not be able to afford it to attend a EuroBSDcon.

The grant covers travel expenses, hotel accommodations, and the conference and social event fees.

If you are interested in attending next year's conference and require financial support, please consider applying for this grant on the EuroBSDcon Foundation website.

The EuroBSDcon Foundation is committed to transparency, and will publish balance sheets and profit and loss statements for 2014 and 2015 on its website in the coming weeks. Statements for 2016 will be published as well, in early 2017.

See you in Paris at EuroBSDcon 2017! ●

Beat Gätzi is a FreeBSD ports committer and a former member of the FreeBSD ports management team. He was part of the program committee for EuroBSDcon 2013 and joined the EuroBSDcon Foundation board in 2014.

Kristof Provost is a freelance embedded-software engineer and FreeBSD src committer. His interests include pf and the network stack at large. When not at a BSD conference he lives in Grimbergen, Belgium.

LINKS

- EuroBSDcon: https://www.eurobsdcon.org
- EuroBSDcon Foundation: https://eurobsdconfoundation.org
- Paul Schenkeveld Travel Grant: https://eurobsdconfoundation.org/paul-schenkeveld-travel-grant/

SEE TEXT ONLY

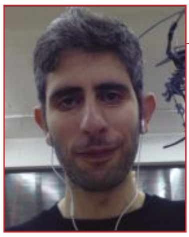
new faces of FreeBSD

BY DRU LAVIGNE

This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community.

This month, the spotlight is on the three committers who joined the Project in September 2016. Sevan Janiyan and Ruey-Cherng Yu became doc committers, and Joseph Mingrone became a ports committer.

Tell us a bit about yourself, your background, and your interests.



Sevan: I'm a 30-something system administrator currently based in London, England. I began working with computers in my early teens, mostly learning through self-study over the years. I have a technical blog called GeekLAN (https://www.geeklan.co.uk) where I write about some of the things I work on.

I actively try to avoid hoarding computers, but somehow my collection of small computers continues to grow. I enjoy playing with different operating systems and listening to electronic music. I would like to learn more about functional and object-oriented programming languages, specifically Lisp and Smalltalk.

Ruey-Cherng: I am a dentist in Taiwan. I didn't major in computer science but I'm very interested in computer science. My father bought an 80286 PC when I was 10 years old. I remembered the OS was DOS 3.3, and I played a lot of DOS games with it. As the hardware requirements went higher and higher, my father upgraded my computer from 80386, 80486DX2-66, Pentium, and so on. After a while, I decided to understand how these games work. In my first year of high school, I attended a C programming language and data structure course in the Department of Computer Science and Information Engineering, National Taiwan University. The compiler I used at that time was Borland C++ 2.0. After finishing the course, I learned that debugging is difficult work so I instead entered medical school. During my college life, I started to know more about the Unix-like operating systems via Redhat Linux 5.2. I also used Mandrake Linux and Debian. My interests are computer science, cycling, economics, and accompanying my two daughters.

Joseph: I am a PhD candidate in the Math/Stats department at Dalhousie University in Halifax, Nova Scotia, Canada, where I work on models of molecular evolution. Aside from doing research, I spend time managing our research group's computing cluster. Before graduate school, I worked as a software developer on a variety of projects using a variety of languages.

How did you first learn about FreeBSD and what about FreeBSD interested you?

Sevan: Long, long ago, I managed to get hold of a Sun3/60 workstation (this was a workstation produced by Sun Microsystems in the 1980s which used a Motorola 68000 CPU) and was looking for an operating system to run on it. I became aware of the BSDs through this effort but really didn't know what was what. I certainly visited the websites of the three major BSD variants some time around 1997.

The first time I actually tried FreeBSD I remember clear as day. I saw the 5.0 release announcement on the news site Slashdot the night it landed. Same situation again: I'd obtained another aging piece of hardware, this time a DEC AlphaStation 250. Linux hadn't worked out so well on this system, and I saw the Alpha CPU was a supported architecture, so I gave FreeBSD a try and wasn't disappointed. The first thing was that my system was way faster than the previous operating Linux distributions I'd tried and secondly, thanks to the Handbook, I was able to achieve a great deal with the system without prior knowledge. This was very empowering and I was hooked.

Ruey-Cherng: When I used Redhat 5.2, I found the dependency problems to be so annoying. This improved after I switched to Debian. When dealing with the dependencies of RPM files and searching for solutions on the Internet, I found one of the features of FreeBSD: the ports system. The convenience of ports interested me. But at that time, the hardware compatibility was not so good, so I just tried FreeBSD for a short period. In May 2014, FreeBSD attracted me again. I installed FreeBSD 9.0 on my Fujitsu T2010 and FreeBSD worked well with some tweaks.

Joseph: I first learned about FreeBSD around 1998. Two roommates and I were moving into an apartment and we hoped to share an Internet connection. This was before cheap consumer routers were common, and I recall the ISP either charging extra for, or forbidding multiple computers on, the same connection. At that point, I had little Unix experience, so I made a deal with an older, wiser computer science student. He chipped in with technical advice and I took extra turns driving for our summer-job carpool. We found a discarded desktop, threw in a second network card, and installed some Linux distribution. I don't recall the details, but I remember it did not work well. This older, wiser student was friends with a FreeBSD alumnus, Marc Fournier, and Marc recommended FreeBSD. After installing FreeBSD, that old desktop ran wonderfully all year. So, FreeBSD was introduced to me and what initially interested me was the stability. Today, there are many reasons why I still run it on servers, desktops, and laptops, but I will highlight one. I appreciate the control FreeBSD gives me. With the relatively minimal base system and powerful ports system, I can customize the installation, unlike any other operating system that I am aware of. Naturally, I am excited about the upcoming packaged base.

How did you end up becoming a committer?

Sevan: I believe I reached the sufficient level of patch submissions to move up to the next level where I get to commit my own proposed changes after being reviewed and given the all clear by my mentor, Benedict Reuschling.

It all began while I was reviewing the source history and documentation for the utilities shipped in the base systems of the current BSDs. I came to the realization that there was an inconsistency regarding the origin of the utilities and which version of operating system they first appeared in. The modern BSDs are in a fortunate position where the lineage can be traced all the way back to 1BSD. The FreeBSD Project hosts a copy of the CSRG repo which can be checked out or browsed with a web browser at https://svnweb.freebsd.org/csrg/. If preferred, the repository along with the actual binaries are available as a CD set from Kirk McKusick for purchase at http://www.mckusick.com/csrg/index.html. With a copy of the CSRG repo and various other resources online to cover other unices, I set out to review the history section of man pages which included it and to add the section to those which were missing it. The result is that the tools in `/bin and /sbin` are now documented. The `/usr/bin` and `/usr/sbin` tools are next.

Ruey-Cherng: When gathering the information I needed to install FreeBSD 9.0, I found a lack of traditional Chinese documentation. The information I did find was outdated, mostly for FreeBSD 5.X. The latest update of the FreeBSD Handbook in zh_TW was in 2008. There was no official FreeBSD website link to the zh_TW version because its content was too outdated. So I decided to update the translation of the FreeBSD Handbook.

I sent an email to the freebsd-doc mailing list to introduce myself and seek help and recommendations. My co-mentor lwhsu@ contacted me and helped me to convert old BIG-5 encoding to UTF-8. Then, I began the translation. delphij@ helped me to solve the mojibake problem in PDF rendering. I set up a FreeBSD server on my Raspberry Pi 2B so that I can ssh to my server from the dental clinic and do some translation work when I have free time. The FreeBSD arm package repository was not set up at that time and I encountered some problems when installing the programs I often use. I met my mentor kevlo@ in the FreeBSD Taiwan User Group on Facebook. He answered lots of my stupid questions kindly and patiently. He solved my problems and encouraged me to continue my translation.

The progress of my work was very slow as I struggled with the xml tags. Fortunately, the evening when I tested and practiced the po translation, I encountered some problems in po translation to UTF-8. My co-mentor wblock@ kindly helped me solve them. The conversion of the old translation method to the po translation is a labor-intensive work. You have to copy and paste all the strings one by one to a po file. It needs to involve more people in the project. To attract more new blood, I gave a talk about the traditional Chinese translation of the FreeBSD documentation at a local BSDDay in Taipei in July (https://lists.freebsd.org/pipermail/freebsd-translators/2016-August/000141.html). For people who could not attend, I also wrote an article to introduce the traditional Chinese translation of FreeBSD documentation. A contributor, cwlin, contacted me via the subversion log. He converted the handbook in zh_TW to po translation which was a very big work! After, kevlo@ proposed me to be a committer to facilitate my translation work. Now with the help of kevlo@ and ryusuke@, I have revived the FreeBSD website in zh_TW.

Joseph: Like most committers, I just started submitting patches. I met FreeBSD people by lurking on IRC and the mailing lists and attending BSDCan. The hacker lounge at BSDCan was a nice way to make connections with Free-, Open-, and even a few DragonFlyBSD community members. Those connections sparked my motivation to contribute more.

I have worked on a variety of ports from games/voxelands to security/wpa_supplicant, but one area where I hope to make more commits could be described as "scientific software." A few examples are biology/diamond, biology/njplot, biology/paml, cad/gmsh, and math/R.

How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

Sevan: I'm very happy to be a member of the Project. The on-boarding process is very easy and fellow team members are very helpful with tips and suggestions. The workflow goes from Bugzilla, which is our bug tracker where requests and patches land, to Phabricator, which is the review system. I really like the review system as it's reassuring to have someone go over your changes. Sometimes things are overlooked and it's good to catch them before they land in the source tree. This process also helps with the team becoming familiar with each other's approach and it provides different perspectives to a problem.

Every year the BSD community gathers in Ottawa, Canada, at a conference called BSDCan. Each evening, there is a documentation sprint. At the doc sprints, folks from the documentation team give talks about various aspects of working on documentation and help people get started with working on the documentation. Unfortunately, BSDCan is once a year so for our day-to-day interaction we meet and hang out on IRC and the mailing lists. There is a wiki article for the IRC networks we are present in: https://wiki.freebsd.org/IRC/Channels#EFnet. #bsddocs on EFnet is where the documentation team can be found as well as on the freebsd-doc@ mailing list.

To someone interested in becoming a FreeBSD committer, I would suggest joining the IRC channel and mailing list relevant to the area of interest. This will get you in touch with other developers who can provide guidance and a helping hand if stuck. If you're unsure of a project to begin working on, a review of the system is always a safe start. This has the benefit of familiarizing you with the current state which can help clarify what new improvements could be made, reducing the likelihood of potential oversights.

Ruey-Cherng: You could install FreeBSD and do your daily work with it. As you find some bugs in the system, ports or documentation, send a bug report with your patch. I installed FreeBSD plus the Xfce desktop environment on my Macbook Pro 2011 to do routine jobs such as surfing the web, word processing, and instant messaging on FreeBSD every day.

The first time I browsed the FreeBSD forums, I thought it a little deserted. I went to browse the mailing lists archive and subscribed to some mailing lists. I found most FreeBSD developers like the old school discussion method of mailing lists. Subscribe to the mailing lists of the fields you are interested in. Soon you will understand why they like mailing lists instead of the web forum as it is very convenient to browse and reply with your favorite email client. Now that you know where the developers gather, introduce yourself and participate in the discussion.

Related bug reports will also be sent to the mailing lists and you will see your bug reports and patches in the mailing lists. The developers will recognize your contributions and provide suggestions.

Look for a local FreeBSD user group and join it. This is a good way to receive new information and solutions for the problems you encounter. You may also meet some developers in the local user group.

Read the FreeBSD Handbook and the other articles and books in the FreeBSD documentation project to learn more about FreeBSD. If you find bugs or outdated information in the documentation, fix the bugs and contribute your updates.

Surf the FreeBSD website in your language and find what you can do to update the content of that website to sync it with the English version.

If you have hardware that FreeBSD doesn't support yet, don't be disappointed. You may have an opportunity to be involved in testing the driver! Seek the driver solution in the related mailing lists, look to see if there is a latest and unreleased driver and help the developers test it.

All your efforts will be appreciated. If you see others discussing issues "loudly," don't be afraid. It is just a smaller society like the real world.

Joseph: My experience has been quite positive. I have fantastic mentors (swills and AMDmi3) who help to steer me in the right direction when I veer off course. Some other, more senior ports developers (especially mat) have offered lots of helpful hints as well. To anyone thinking about contributing to the ports tree, I would say, with over 25,000 ports, we need you! The Porter's Handbook and tools like poudriere and portlint make chipping in accessible, even for people with less experience. •

DRU LAVIGNE is a doc committer for the FreeBSD Project and Chair of the BSD Certification Group.

Thank you!

The FreesBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.

FreeBSD™ FOUNDATION

Are you a fan of FreeBSD? Help us give back to the Project and donate today! **freebsdfoundation.org/donate/**

Please check out the full list of generous community investors at freebsdfoundation.org/donate/sponsors

Uranium

The Koum Family

Anonymous

Iridium

NetApp

Platinum

Juniper Networks

Gold

ARM facebook NETFLIX

Silver

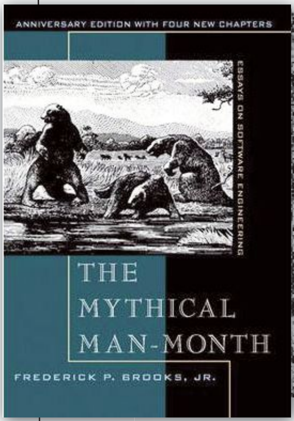
accelerationsystems STORMSHIELD

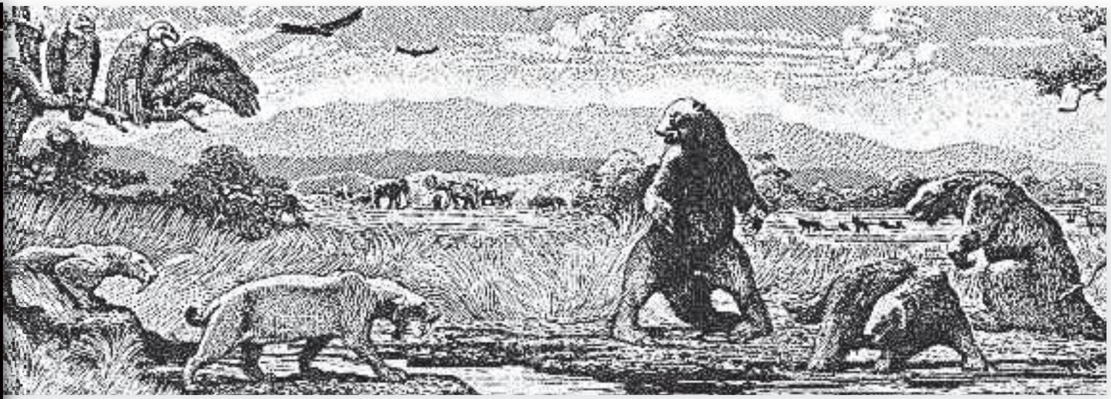
Google iXsystems

Tarsnap vmware

SEE TEXT ONLY

BOOKreview by John Baldwin


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.



The Mythical Man-Month and Its Relevance to the FreeBSD Project

The Mythical Man-Month	*By Frederick P. Brooks Jr.*
Publisher	*Addison-Wesley Professional; Anniversary edition (1995)*
Print List Price	*$42.99*
Digital List Price	*$34.99*
ISBN-10	*0201835959*
ISBN-13	*978-020185953*
Pages	*336*

The Mythical Man-Month is a classic work on software development. It is structured as a series of essays that use anecdotal evidence as the basis for practical advice. Rather than dealing with specific technical details, these essays focus on how people develop software.

I first read *TMM* (as it is otherwise known) less than a year removed from college graduation on the recommendation of FreeBSD developers. While some aspects of programming practice have changed since *TMM* was originally written in 1975 (such as widespread use of higher-level programming languages and interactive debugging), many of the concerns the author, Fred Brooks, notes are still relevant today.

One of the first topics Brooks broaches is distinguishing a simple computer program from a product. A developer who bangs out a program to accomplish a task for himself or herself has written a computer program. A product, on the other hand, must be usable by other people. It must handle arbitrary inputs (including erroneous inputs) correctly. It must have documentation that explains both how to use the program and which tasks a program is suitable (or not suitable) for. Developing a product requires far more effort than the time to implement the initial computer program. In FreeBSD, programs should have a manual page at a minimum. However, supplemental documentation in the Handbook is often required (and not always present).

Similarly, designing a computer program to be a part of a computer system requires significant work. Programs belonging to a system should provide consistent interfaces to users. They should also provide consistent methods for cooperation. One of the ways UNIX-like systems achieve this is through the use of pipes to join the input and output streams of processes. However, there are several other ways to achieve consistency, such as reusing the same option letters for common command line flags or using consistent language in manual pages.

Chapter 2 discusses the scheduling of software projects. In particular, it seeks to destroy the notion that adding developers to a software project increases productivity (and reduces schedule length) linearly. Instead, Brooks highlights the communication overhead imposed when adding developers to a project as well as the additional time required for training. Brooks also notes that many software projects contain a critical path of tasks that must be completed in sequence rather than in parallel. These hard truths are used to warn against overly optimistic responses to schedule slippage, and in particular, the knee-jerk reaction of adding additional developers to a software project that is behind schedule.

FreeBSD releases do not have a combination of a hard deadline along with a required set of features in quite the same way Brooks's examples imagine. However, several of Brooks's observations are still applicable. Even with the ability to drop features and focus on just shipping a release on a given date, FreeBSD releases (especially the X.0 releases on new branches) are often delayed. These delays usually stem from critical path tasks such as waiting for a pending security advisory scheduled for release near the time of the release. FreeBSD also contains a large developer base, which results in a large portion of time spent on communication.

Chapter 3 proposes a model of organizing developers into teams similar to the organization of teams used for medical surgeries. Each team consists of a single "surgeon" who is tasked with writing the code for the end product as well as additional team members to assist the surgeon. FreeBSD developers are not generally organized in teams as proposed by Brooks. However, Brooks's model does highlight tasks that are important to development beyond simply writing code. Two of the team members he calls for are dedicated to tools and testing.

The FreeBSD community has expended a significant level of effort on improving the tools available on FreeBSD in the past decade. These include adopting more modern replacements for toolchain components such as Clang, as well as entirely new tools such as DTrace. This work continues with improvements to debuggers and performance analysis utilities.

FreeBSD developers have also increased the testing coverage. For several years, FreeBSD's source tree included a hodgepodge of tests that were not run regularly. Some tests used common frameworks, but many other tests were standalone. Through the work of several developers, FreeBSD adopted the kyua testing framework and first shipped a set of kyua-based tests in 10.1-RELEASE. Many existing tests from NetBSD were imported as part of the process, and existing tests were converted to the new framework. These tests can now be easily run as automated jobs. There is still a lot of work to do on this front, not only with adding tests to fill in gaps in existing test suites, but new classes of tests such as tests of the installer.

Chapter 4 highlights the importance of conceptual integrity and offers a plan for achieving it in large systems. Brooks asserts that "conceptual integrity is the most important consideration in system design." He states that the only way to achieve this is to limit the number of minds involved in the design. Brooks proposes splitting architecture from implementation. This concentrates the design in the minds of the architects while spreading the workload of implementation across a larger pool of developers. FreeBSD does not place a sharp distinction between these roles and does not have any formal, system-wide architects. Informally, FreeBSD developers do seek review from peers, and there are individuals whose review is sought for changes to specific portions of the system. In some cases, these individuals are de-facto architects for those portions of the system, but not formally named. FreeBSD has attempted to make this process more formal in the past, but these attempts have not succeeded.

The Second-System Effect is the topic of Chapter 5. Here, Brooks documents developers' tendencies to pare down features in the first version of a system only to swing to the opposite extreme in the second system. Having successfully completed the initial version, the developer then proceeds to add every conceivable feature to version 2.0. Brooks states this system "is the most dangerous system a man ever designs." The result is a bloated system with poor conceptual integrity. FreeBSD is not a perfect system and certainly has had portions of the system that have suffered from this tendency. Our developers are continually learning from each other as well as from our own past mistakes. Sometimes our peer review permits us to fix mistakes sooner rather than later, but not always. However, FreeBSD's developer community provides a place where developers can be mentored and warned about trends such as the Second-System Effect.

Chapters 6 and 7 highlight the importance of effective communication and organization as requirements for effective development involving a group of individuals. FreeBSD developers and community members communicate over various venues including email, web forums, and real-time chat such as IRC. However, our community also realizes the value of in-person communications and places a premium on "face time" at conferences, developer summits, and user group meetings. The community actively participates in these meetings not just as attendees, but as organizers, speakers, and volunteers. Organization remains a challenge for FreeBSD. The FreeBSD Project's formal membership is very developer-heavy, but is in need of a variety of skill sets apart from pure coding. This is not to say that the Project is completely disorganized, but there are certainly gaps in the coverage of many of the non-coding tasks the Project requires for maximum effectiveness.

Estimating the time and resources needed for systems projects is the subject of Chapter 8. Brooks cites results from various studies analyzing some of the factors that affect scheduling of systems projects. One of the first studies highlights the fact that a significant portion of programmers' working days are lost to factors such as meetings, working on tasks for other projects such as a high-priority bugfix, or hardware failures. Other studies note that the lines of code produced over time vary with the complexity of the program. Of the projects surveyed, operating systems were the most complex (and thus the slowest to develop).

Chapter 9 deals with managing memory allocation between portions of a large program. From a technical standpoint, much of this chapter has been made obsolete by virtual memory. However, the chapter still contains a few nuggets. First, Brooks relates an anecdote from the OS/360 development that resulted in the system performing very poorly. Each module was assigned a size constraint and the developers of each module resulted to workarounds such as overlays and "borrowing" space from neighboring modules without considering the effect on the system at large. While each module functioned in isolation within its assigned space, the system as a whole suffered from excessive disk I/O to satisfy overlay requests. Among several lessons from this story is the point that one must keep system-wide effects in mind and not focus exclusively on one module. A second point made in this chapter is the importance of data representation on performance. The data representation often drives the algorithms constraining the range of performance. In those cases, large performance changes will come through altering the data representation rather than optimizing the existing code flow.

Just as it is important to plan for change in the design of software, it is also important to plan for design in the organization building the software. For FreeBSD, this means building and maintaining a community, not just source code.

In Chapter 10, Brooks discusses the importance of formal documents and their use in setting down design. In particular, deriving specifications requires one to consider a host of design decisions that would otherwise not be realized until deep in the bowels of a coding session. This is certainly an area where the FreeBSD Project is not very strong. As an open-source project, FreeBSD's developers are not handed a set of specifications from external customers to build a design against. In many cases, FreeBSD's developers are themselves FreeBSD consumers and apply their own design requirements to FreeBSD. In addition, the Project actively seeks feedback from our customers to determine their requirements. While we are not always able to satisfy every requirement, we do use these to set Project direction. Vendor summits are one of the tools the Project uses to engage with our customers. These are often held in conjunction with an existing conference or developer summit. If you are a FreeBSD customer looking to engage with the Project, please contact the author or another FreeBSD developer. We want to hear from you.

Chapter 11 focuses on change. Requirements of a software product change over time as does the hardware software runs on. Brooks encourages developers to embrace change and plan for it rather than fighting against it. For new software projects (or even new subsystems), one often needs to build an initial prototype to better understand the problem being solved. Just as deriving specifications forces consideration of several design decisions, implementing a prototype also uncovers a host of unforeseen issues. Trouble arises when one assumes that this initial version must ship as the final product rather than having the freedom to re-architect the design of the actual product. Just as it is important to plan for change in the design of software, it is also important to plan for design in the organization building the software. For FreeBSD, this means building and maintaining a community, not just source code. Over its lifetime, the FreeBSD Project has benefited from the contributions of many individuals. Often individuals move on to other interests or tasks after working on FreeBSD for a period of time. FreeBSD has survived these fluctuations in our community and has continued to grow. However, we must continue to actively welcome and recruit new individuals to our community.

The importance of tools is the subject of Chapter 12. While the tools-smith from Chapter 3 is mentioned, this chapter focuses on other topics than programming tools. In particular, Brooks discusses the trade-offs of sharing access to new hardware among multiple developers, the importance of simulators relative to actual hardware, and the benefits of interactive debugging. Interactive debugging is something that developers now take for granted, but the other two topics remain prescient today. Bringing up FreeBSD on the arm64 platform has encountered constraints with target machine scheduling due to the limited number of machines available. Simulators such as QEMU also permit developers to test FreeBSD on a broader range of platforms from the comfort of their desktops.

Brooks next turns to system debugging in Chapter 13. The chapter opens with an emphasis on architecture and design. Skimping on the planning and design stages sows the seeds of an unstable and buggy system. Brooks favors a top-down design approach using iterative refinement to bring detail to the design. Much of the chapter warns against various shortcuts during system test (now referred to as integration testing). First, test and debug individual components in isolation rather than attempt to test multiple components together. The first approach does require more "scaffolding," but in the latter approach, bugs in components can interact in surprising ways resulting in breakage that is much harder to debug. In some cases, multiple bugs across components may cancel out each other in a larger test, giving a false sense of correctness. Second, don't skimp on the extra code and tools that are needed for testing, but not in the final product. These enable more detailed component testing which saves time and frustration later in integration testing. Third, maintain a standard copy of the current system that new and in-progress components are tested against. Changes to this system should be rolled out periodically in a schedule that provides periods of stability for in-progress components to be tested against. Finally, new components should be added to the system one at a time, and the new system should run a full complement of regression tests as each component is added.

While Chapter 8 covered the topic of planning and estimating for a software project, Chapter 14 focuses on managing a project once it is in progress. Brooks's first recommendation is to have a schedule that contains concrete milestones. Vague milestones result in ambiguous communication between layers of management (or between managers and developers). Brooks also warns against micromanaging. When managers jump in to correct issues the managees are capable of solving, the managees resort to hiding reports of issues as a means of self-defense. To foster clear communication, a manager must be willing to accept bad news without immediately barking orders.

Chapter 15 addresses the subject of program documentation. Brooks begins by noting three levels of documentation required for a program: how to use a program, how to test a program, and how to modify a program. Brooks states that one of the root problems at the time of writing was that documentation was kept separate from source and inevitably became stale. He proposes self-documenting source code as the most viable solution to this problem. Self-documenting source is now widely practiced, and in particular, the use of high-level languages has made this more reasonable. In-source documentation systems such as Javadoc and Doxygen have been used by many software projects to store documentation alongside the source that can be easily translated into a more human-readable format. While certain portions of FreeBSD's kernel source do use Doxygen comments, most of FreeBSD's documentation for the first two levels is stored separately from the source. Some documents are not well suited to storing in source, but API descriptions may be better served by a system like Doxygen rather than stand-alone manual pages.

One of the documentation topics Brooks mentioned while describing the last level of documentation was to explain "why" design decisions were made. While these decisions are sometimes discussed in comments in the source code, these decisions are often explained in the source code repository logs of modern software projects. FreeBSD (and BSD) have a long history of using source code control, and the FreeBSD Project culture expects and encourages thoughtful log messages that explain the "why" of changes, not just the "what."

The 20th-anniversary edition of *TMM* includes an essay titled "No Silver Bullet" as Chapter 16. The essay was originally published in 1987 and discusses software productivity. In particular, Brooks asserts that in the following decade (1987–1996), no "silver bullet" would arise to improve the productivity of software developers by an order of magnitude. Brooks begins by dividing the difficulties of software development into two classes: essence and accidents. The essence of software is its abstract design. It includes the representation of data and algorithms used. Accidents deal with difficulties in expressing this abstract design. These can include the limitations of hardware and the languages and forms used to express an idea.

Brooks claims that the three large steps in improving software productivity prior to 1987 all attached accidental difficulties. High-level languages permit programmers to more concisely express concepts while deferring many of the mundane details to the compiler. Time-sharing systems facilitate quick turn-around time during development, allowing developers to sustain concentration for long periods of time. Unified programming environments provide a standard way to connect existing programs together that can be used to solve larger tasks, such as using an I/O pipeline in UNIX.

The essence of software development remains hard. Software is complex. Brooks notes that in many other objects people create, repeated elements are common, but in software, a repeated element is consolidated into a subroutine. Newer versions of software projects are not formed solely by duplicating existing modules. Instead, they contain entirely new components that must interact with the existing modules. At the same time, software is invisible and its structure defies detailed visualizations. When visualizations are used, they are forced to constrain the information, by focusing on control-flow rather than data-flow, for example.

Brooks asserts that difficulties arising from this essence are inherent in software. He did not see any techniques proposed or in practice in 1987 that would attack these difficulties in a substantial way over the ensuing decade.

In Chapter 17 Brooks responds to some of the criticisms of "No Silver Bullet" and critiques additional silver bullet candidates nine years after the publication of the original paper.

To encourage more vigorous debate of the various propositions in the original *TMM*, Chapter 18 includes a bulleted list.

Finally, in Chapter 19 Brooks revisits the content of the first 15 chapters 20 years after they were first written. In some ways, this is the most interesting chapter of the book. Brooks notes topics made obsolete by technological advances while affirming the propositions he believes are still valid. Certainly the power of *The Mythical Man-Month* stems from its continued relevance decades after it was first written. In part, this is due to the essence of software described in "No Silver Bullet." Technological advances have not altered the fundamental construct of software. Secondly, *TMM* is largely about people and the interactions of people who work with software. While new technology does affect our lifestyle and the "accidents" of our task, people are still people.●

JOHN BALDWIN (jhb@freebsd.org) joined the FreeBSD Project as a committer in 1999. He has worked in several areas of the system including SMP infrastructure, the network stack, virtual memory, and device driver support. John has served on the Core and Release Engineering teams and organizes an annual FreeBSD developer summit each spring.

Testers, Systems Administrators, Authors, Advocates, *and of course* **Programmers** *to join any of our diverse teams.*

COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ DOWNLOAD OUR SOFTWARE ★

http://www.freebsd.org/where.html

★ JOIN OUR MAILING LISTS ★

http://www.freebsd.org/community/mailinglists.html?

★ ATTEND A CONFERENCE ★

Linux.conf.au • Hobart, Tasmania, Australia • January 16–20, 2017
FOSDEM 2017 • Brussels, Belgium • February 4–5, 2017
AsiaBSDCon 2017 • Tokyo, Japan • March 9–20, 2017

The FreeBSD Project

FreeBSD FOUNDATION

SEE TEXT ONLY

conference REPORT™

by Samantha Bonham

MeetBSD 2016

The BSD operating system first came to life at the U.C. Berkeley campus in the 1970s, and this year I had the chance to meet FreeBSD enthusiasts from all walks of life at the place where it all began. MeetBSD 2016 did not follow the typical conference format; instead, it was a mix of formal talks, group discussions, and breakout sessions, which made it the perfect conference for someone like me, who is new to the FreeBSD community and still learning the technical side of things.

The Day Before the Conference

On the day before the conference, our Documentation Team decided to meet at the Berkeley campus for a bit of a "hacking session." It was a little over an hour's drive before I arrived at the campus, significantly less travel time than the majority of my team members who flew in from Tennessee.

While the drive was relatively short and full of scenic views, it was not without its challenges. If you are new to driving stick shift, I highly recommend avoiding driving anywhere near the U.C. Berkeley area, and travelling by train instead. One-way streets, hordes of student pedestrians, and stoplights on hilly roads will make even the most seasoned stick shift driver a little bit twitchy.

Google Maps took me on a slight diversion, as it kept telling me to turn left on a one-way street, but despite my lack of navigational skills, I did finally make it to the Clark Kerr campus at around 3 p.m. The lush greenery and quiet atmosphere at the campus offered the right amount of Zen to counteract my travels. Once I met with the team, we went straight into an outdoor hacking session. This involved playing musical chairs with power outlets, gigantic mutant spiders, testing TrueOS on a virtual machine, and hacking a Raspberry Pi which was missing some parts. After an hour-long hacking session, we decided it was time to check into the hotel before going out to dinner. At 7 p.m., a group of us strolled down Durant Avenue in search of food, and after seriously considering eating donuts for dinner, we ended up having a nice meal at the hotel.

First Day of the Conference

After a six-minute ride with Uber, we arrived for the first day of the conference at 9 a.m. The first formal talk was scheduled for 10 a.m., so we had plenty of time to mingle, drink coffee, and munch on an

assortment of tasty pastries. We also had time to collect our swag bags, and I ended up making a donation for a FreeBSD Foundation thermos, which kept my coffee warm throughout the conference.

The scene at the conference hall was that of a typical college lecture, except the "students" had better laptops and an affinity for daemon horns. The conference format was split into 45-minute segments, with talks covering a variety of FreeBSD-related subjects, ranging from jail networking to the History of ZFS and OpenZFS. After each talk, there was 10 minutes for questions.

The first talk was given by Devin Teske, who did a live demonstration on creating and managing `vimage` jails. During her talk, I learned that `netgraph` and `if_bridge` can be used to create multiple jails with with proper network configuration. I also found out that while it is technically possible to run a ZFS server in a VNET jail, it is not recommended, as systems have been known to kernel panic when there is an active NFS mount.

In the afternoon, there was a "Topic Brainstorm Session," where we were given the link to a Google Doc and asked to share topics we wished to discuss. About six topics were shared, but the one that generated the most interest at our table was definitely "Trolling Linux Admins for Fun and Profit."

After we took a break for lunch, Matt Ahrens, one of the cofounders of the ZFS filesystem, walked us through a brief history of ZFS and OpenZFS. During his talk, he touched on key milestones of the project, such as the first implementation of deduplication in 2009 at Sun. He also spoke about the project's future goals, which included performance improvements on all SSD pools, integration with cloud management, object storage management, and fault management.

The conference came to a close at 6 p.m., just in time for a pizza party at the Hillside Club. I guess everyone was really hungry because the pizza ran out quickly. I had fun meeting new people, and the party really started to liven up when the FreeBSD mascot, Beastie, made a guest appearance.

Last Day of MeetBSD

Day two of the conference got off to a great start with a talk by Rod Grimes about the early days of FreeBSD. Having the conference in the Bay Area meant that a lot of old-timers in the FreeBSD community were able to participate in the conference this year. Mr. Grimes was one of them, and his talk was particularly inspiring as he was FreeBSD's first release engineer. Mr. Grimes recently rejoined the FreeBSD release engineering team after a 20-year-plus hiatus.

Later that morning, cofounder of the FreeBSD Project Jordan Hubbard gave a live demo of FreeNAS 10. It is a complete rewrite of the original FreeNAS operating system with an asynchronous user interface and middleware, offering features like Docker support and bhyve VMs. Jordan also touched on the challenges of using FreeBSD to build a modern storage platform. At the end of his talk, he encouraged any interested developers to contribute to the FreeNAS 10 project.

Another great talk was the unveiling of the TrueOS platform, which is the successor to PC-BSD. TrueOS is a desktop or server operating system based on FreeBSD. Founder of PC-BSD, Kris Moore, explained the reasoning behind the name change and discussed some of the new features that have been added to the TrueOS desktop environment, like support for 4K and newer Intel/AMD graphics cards and encrypted home directories. The desktop version of TrueOS is a truly unique experience; I have enjoyed tinkering around with it on my new laptop.

Conclusion

It doesn't matter if you're new to the open-source community or a FreeBSD expert, MeetBSD is an experience that should not be missed. The nontraditional structure of the un-conference and the laid-back attitude of the people combine to create a truly unique way to learn about the vibrant FreeBSD culture. ●

SAMANTHA BONHAM is a technical writer with iXsystems, writing online and contextual documentation to improve the FreeNAS and TrueNAS user experience. Formerly a photojournalist for the Cayman Compass, a daily newspaper in the Cayman Islands, Samantha graduated in 2012 from Sheffield Hallam University in the United Kingdom with a Bachelor of Arts in Communications. Currently living in the Santa Cruz Mountains, she likes to spend her free time on photography, birdwatching, and yoga.

SEE TEXT ONLY

svnUPDATE

by Steven Kreuzer

Now that FreeBSD 11 has been released, we've seen quite a bit of activity in src, which I can only imagine is because developers had changes they wanted to make but didn't want to commit during a release cycle. Now that it's "business as usual," we've seen a flurry of activity across all subsystems. One of the most interesting changes is that both cron and syslog have gained functionality, which makes it easier to roll out changes to these programs using your favorite configuration management system. As the size of infrastructures grows and we move toward the philosophy of cattle, not pets, when talking about the servers we manage, not having to write complex logic to quickly push out changes is something I am sure any system administrator will welcome with open arms. Hopefully this is the start of a trend that other programs in the base system will start to follow as well.

`cron(8)`: add support for `/etc/cron.d` and `/usr/local/etc/cron.d` (https://svnweb.freebsd.org/changeset/base/308139).

Cron jobs can now be broken up into individual files, which makes deploying and managing these jobs using various automation tools significantly easier than having to modify `/etc/crontab`.

`syslogd(8)`: add an '`include`' keyword (https://svnweb.freebsd.org/changeset/base/308160).

All the `'.conf'` files not beginning with a `'.'` contained in the directory following the include keyword will be included. The default `syslogd.conf` has been updated to `'include'/etc/syslog.d` and `/usr/local/etc/syslog.d`.

`daemon(8)`: Allow logging daemon stdout/stderr to file or syslog (https://svnweb.freebsd.org/changeset/base/307769).

The daemon utility detaches itself from the controlling terminal and executes the program specified by its arguments. In the past, any output created by the process would have been written to a file on local disk. This change will allow centralization of the output to a standard location and even shipping it off to a remote syslog server.

Support for the Raspberry Pi 3 (https://svnweb.freebsd.org/changeset/base/307257).

Initial support for the Raspberry Pi 3 has been committed. While SMP, VCHIQ, and the RNG driver are not currently supported, multiple FreeBSD developers continue to hack on at this very popular platform, so keep an eye on the freebsd-arm mailing list to track the progress.

Allow an SMP kernel to boot on Cortex-A8 (https://svnweb.freebsd.org/changeset/base/308213).

This change allows FreeBSD to boot on all ARMv7+ Cortex-A cores with 32-bit support.

Support for Allwinner H3 audio codec (https://svnweb.freebsd.org/changeset/base/308269).

The audio controller in the H3 is more or less the same as A10/A20—except some registers are shuffled around. The mixer interface, however, is completely different between SoCs. Separate `a10_mixer_class` and `h3_mixer_class` implementations have been made available, which makes adding support for other SoCs much easier.

virtio-console support to bhyve (https://svnweb.freebsd.org/changeset/base/305898).

A new device driver has been added to bhyve, which allows the creation of up to 16 bidirectional character streams between host and guest.

`zfsbootcfg(8)` provides the ability to set one-time next boot options for `zfsboot` (https://svnweb.freebsd.org/changeset/base/308089).

(`gpt)zfsboot`, will read one-time boot directives from a special ZFS pool area. The area, previously described as "Boot Block Header", but currently known as Pad2, is marked as reserved and zeroed out on pool creation. The new code interprets data in this area, if any, using the same format as `boot.config`.

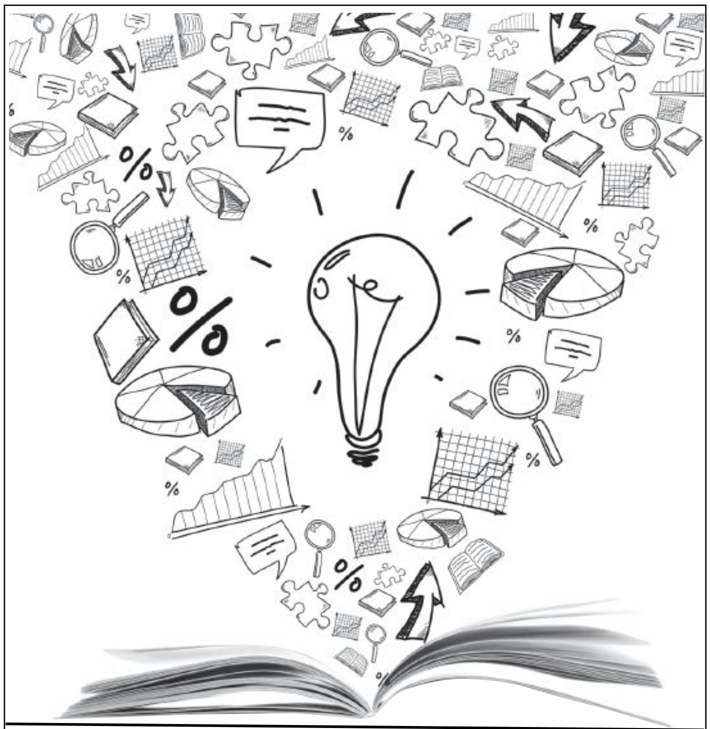
Manipulate EFI variables from userland. (https://svnweb.freebsd.org/changeset/base/307072).

Efivar(1) is a new utility to manipulate Extensible Firmware Interface variables. It has a similar command line interface as the Linux equivalent, as well as adding a number of useful features to make it easier to use in shell scripts.

A new MACHINE_ARCH has been introduced for Freescale PowerPC e500v2 cores (https://svnweb.freebsd.org/changeset/base/307761).

The Freescale e500v2 PowerPC core does not use a standard FPU. Instead, it uses a Signal Processing Engine (SPE)—a DSP-style vector processor unit, which doubles as an FPU. The PowerPC SPE ABI is incompatible with the stock powerpc ABI, so a new MACHINE_ARCH was created to deal with this.

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.



FreeBSD™ JOURNAL™

WRITE FOR US!

Contact Jim Maurer (jmaurer@freebsdjournal.com) with your article ideas.

THROUGH MARCH 2017 BY DRU LAVIGNE

2017 Events Calendar

The following BSD-related conferences will take place in the first quarter of 2017. More information about these events, as well as local user group meetings, can be found at www.bsdevents.org.

FOSDEM • Feb 4 & 5 • Brussels, Belgium

https://fosdem.org/2017/ • FOSDEM is a free event that offers open-source communities a place to meet, share ideas, and collaborate. This annual event attracts over 5,000 attendees each year. This year's event features a BSD devroom, a FreeBSD booth in the expo area, and an opportunity to take the BSDA certification exam.

SCALE • Mar 2–5 • Pasadena, CA

https://www.socallinuxexpo.org/scale/15x/ • The 15th annual Southern California Linux Expo will once again provide several FreeBSD-related presentations and a FreeBSD booth in the expo area. This event requires registration at a nominal fee.

AsiaBSDCon • Mar 9–12 • Tokyo, Japan

http://2017.asiabsdcon.org/ • This is the annual BSD technical conference for users and developers on BSD-based systems. It provides several days of workshops, presentations, a Developer Summit, and an opportunity to take the BSDA certification exam in either English or Japanese. Registration is required for this event.